Scalable Monitoring Technique for Detecting Races in Parallel Programs*

Yong-Kee Jun[1]** and *Charles E. McDowell*[2]

[1] Dept. of Computer Science, Gyeongsang National University
Chinju, 660-701 South Korea
jun@nongae.gsnu.ac.kr

[2] Computer Science Department, University of California
Santa Cruz, CA 95064 USA
charlie@cs.ucsc.edu

Abstract. Detecting races is important for debugging shared-memory parallel programs, because the races result in unintended nondeterministic executions of the programs. Previous on-the-fly techniques to detect races have a bottleneck caused by the need to check or serialize *all accesses* to each shared variable in a program that may have nested parallelism with barrier synchronization. The new scalable monitoring technique in this paper reduces the bottleneck significantly by checking or serializing *at most* $2(B+1)$ *non-nested accesses in an iteration* for each shared variable, where B is the number of barrier operations in the iteration. This technique, therefore, makes on-the-fly race detection more scalable.

1 Introduction

A race is a pair of unsynchronized instructions, in a set of parallel threads, accessing a shared variable where at least one is a write access. Detecting races is important for debugging shared-memory parallel programs, because the races result in unintended nondeterministic executions of the programs. Traditional cyclical debugging with breakpoints is often not effective in the presence of races. Breakpoints can change the execution timing causing the erroneous behavior to disappear.

On-the-fly race detection instruments either the program to be debugged [1,3, 5], or the underlying system [8,12,13], and monitors an execution of the program to report races which occur during the monitored execution. One drawback of existing on-the-fly techniques is the run-time overhead which is incurred from the need to check or serialize all accesses to the same shared-memory location. Every access must be compared with the previous accesses stored in a shared data

* University Research Program supported by Ministry of Information and Communication in South Korea.

** In Gyeongsang National University, he is also involved in both Institute of Computer Research and Development, and Information and Communication Research Center, as a research professor.

structure, often called the access history. In addition, the access history must be updated. This overhead has limited the usefulness of on-the-fly techniques.

The overhead can be reduced by detecting only the first races [3, 5, 8, 9, 12], which, intuitively, occur between two accesses that are not causally preceded by any other accesses also involved in races. It is important to detect the first races efficiently, because the removal of the first races can make other races disappear. It is even possible that all races reported by other on-the-fly algorithms would disappear once the first races were removed. A previous paper [5] presents a scalable on-the-fly technique to detect the first races, in which at most two accesses to a shared variable in each thread must be checked. However, this technique is restricted to parallel programs which have neither nested parallelism nor inter-thread synchronization.

In this paper, we introduce a new scalable technique for programs which may have nested parallelism and barrier synchronization. After first describing the background information on this work, in Section 3 we introduce a set of accesses, called *filtered accesses*. The set includes any accesses involved in first races. We then introduce two filtering procedures which examine if the current access is a filtered access during the execution of a program. Checking only filtered accesses is sufficient for detecting first races in the execution instance and reduces the number of non-nested accesses in an iteration that must be checked or serialized to *at most* $2(B + 1)$ for each shared variable, where B is the number of barrier operations in the iteration. Before concluding the paper, we briefly mention some related work in section 5.

2 Background

This work applies to shared-memory parallel programs [10, 11] with nested fork-join parallelism using parallel sections[1] or parallel loops. In this paper we use `PARALLEL DO` and `END DO` as in PCF Fortran [11]. The program may have inter-thread coordination in the loops using barriers. The *nesting level* of an individual loop is equal to one plus the number of the enclosing outer loops, and each loop may enclose zero or more disjoint loops at the same level. For example, Figure 1 shows a parallel loop of nesting depth two, which has two loops in the second nesting level.

In an execution of the program, multiple threads of control are created at a `PARALLEL DO` and terminated at the corresponding `END DO` statement. These fork and join operations are called *thread operations*. The concurrency relationship among threads is represented by a directed acyclic graph, called a *Partial Order Execution Graph (POEG)* [1]. A vertex of a POEG represents a thread operation, and an arc originating from a vertex represents a thread starting from the corresponding thread operation. Figure 1 shows a POEG that is an execution instance of the program shown in the same figure, where a small filled circle on a thread represents an access executed by the thread to shared variable X. If the program contains barrier synchronization, the POEG will contain additional edges to reflect the induced ordering.

[1] The work in this paper also can be applied to parallel sections without difficulty.

```
...
PARALLEL DO I = 1, 2
  ··· = X                 {r0, r8}
  IF ··· THEN
    PARALLEL DO I = 1, 2
      ··· = X             {r1, r2}
      ··· = X             {r3, r4}
    END DO
  ··· = X                 {r5, r9}
  IF ··· THEN
    PARALLEL DO I = 1, 2
      IF ··· THEN ··· = X {r6}
      IF ··· THEN X = ··· {w10}
      X = ···             {w7, w13}
    END DO
  X = ···                 {w11, w14}
  IF ··· THEN ··· = X     {r12}
END DO
...
```

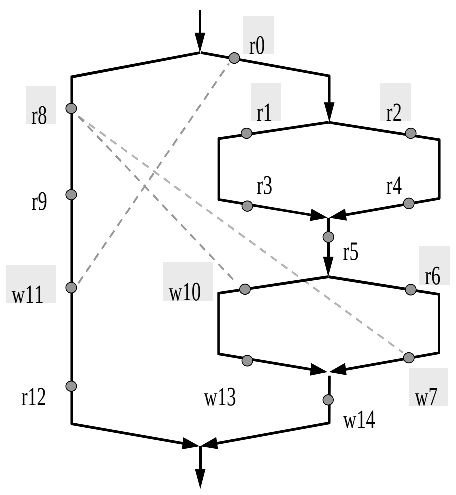

Fig. 1. A Parallel Program and Partial Order Execution Graph

Because the graph captures the *happened-before* relationship [6], it represents a partial order over the set of events executed by the program. Concurrency determination is not dependent on the number or relative execution speeds of processors executing the program. An event e_i *happened before* another event e_j if there exists a path from e_i to e_j in the POEG, and e_i is *concurrent* with e_j if neither one happened before the other. For example, consider the accesses in Figure 1, where $r0$ happened before $w7$ because there exists a path from $r0$ to $w7$, and $r0$ is concurrent with $w11$, because there is no path between them.

Definition 1. *A* dynamic iteration *of a parallel loop in the i-th nesting level, called a* level i iteration *consists of three components: (1) a thread, t_f, in the i-th nesting level that starts immediately at a fork operation, (2) a thread, t_j, in the i-th nesting level that terminates at the corresponding join operation, and (3) a set of threads, T, such that t_f happened before T which happened before t_j.*

Two accesses to a shared variable are *conflicting* if at least one of them is a write. If two accesses, a_i and a_j, are conflicting and concurrent then the two accesses constitute a *race* denoted a_i-a_j. An access a_j is *affected* by another access a_i, if a_i happened before a_j and a_i is involved in a race. A race a_i-a_j is *unaffected*, if neither a_i nor a_j are affected by any other accesses. The race is *partially affected*, if only one of a_i or a_j is affected by another access. A *tangle* T is a set of partially affected races such that if a_i-a_j is a race in T then exactly one of a_i or a_j is affected by a_k such that a_k-a_l is also in T. A *tangled race* is a partially affected race that is in a tangle.

Definition 2. *A* first race *is either an unaffected race or a tangled race.*

There are twenty seven races in the POEG shown in Figure 1; all of the accesses in the POEG are involved in races. Among these, only three races, $\{r0$-$w11$, $w7$-$r8$, $r8$-$w10\}$, are first races which are tangled races. Eliminating the three tangled races may make the other seven affected races disappear. The term tangled race was introduced by Netzer and Miller [9] describing the situation when no single race from a set of tangled races is unaffected by the others. Note that there can never be exactly one tangled race in an execution. They also introduce a tighter notion of first race, called *non-artifact race*, which uses the event-control dependences to define how accesses affect each other.

3 Scalable Monitoring Technique

On-the-fly race detection performs a relatively expensive check each time a monitored access is executed. In the worst case, this check must be done for every access. If we can determine a smaller set of accesses which includes all the accesses involved in first races, then the number of expensive access history operations can be reduced. In this section, we first define such a set, called *filtered accesses*, and then introduce two filtering procedures which examine if the current access is a filtered access during execution of the program. Checking only filtered accesses is sufficient for detecting first races in the execution instance, and reduces the number of expensive checks or serializing accesses to *at most* $2(B+1)$ *non-nested accesses in an iteration* for each shared variable of a parallel loop, where B is the number of barrier operations in the iteration.

Detecting first races in programs with nested parallelism requires detecting the happened-before relationship between nested iterations. We first exploit the nested iterations to indicate a set of filtered accesses.

Definition 3. *A read (write) access, a_i, is a* level k filtered read (write)*, if and only if (1) a_i is in level k iteration I_k, and (2) there does not exist any other access, a_j, such that a_j is in I_k, a_j happened before a_i, and there are no barrier operations on the path from a_j to a_i in the POEG.*

Definition 4. *A write access, w_i, is a* level k filtered r-write*, if and only if (1) a_i is in level k iteration I_k, (2) there exists a level k filtered read, r_i, such that r_i happened before w_i, and (3) there does not exist any other write access, w_j, such that w_j is in I_k, w_j happened before w_i, and there are no barrier operations on the path from w_j to w_i in the POEG.*

For example, consider the accesses in Figure 1. There are five filtered accesses in the two level 1 iterations of the execution instance: two filtered reads $\{r0, r8\}$ and three filtered r-writes $\{w7, w10, w11\}$. Among these accesses, $\{r8, w11\}$ are in the same level 1 iteration which is in the left in the POEG, and $\{r0, w7, w10\}$ are in the other level 1 iteration. The reads, $\{r0, r8\}$, are level 1 filtered reads because there does not exist any other access that happened before $r0$ or $r8$ in their level 1 iterations. The access $w7$ is a level 1 filtered r-write, because there exists a filtered read $r0$ in the first level that happened before $w7$, and there does not exist any other write access that happened before $w7$ in the level 1 iteration. In the second level, we have four iterations in the figure, which have five reads

```
**CheckRead**$(X, k, bv, c\_r)$
**if** $\neg P(X, k, bv_k) \ \wedge \neg F(X, k, bv_k)$ **then**
  **for** $i := 1$ **upto** $k$ **do**
    $P(X, i, bv_i) :=$ **true**;
  **endfor**
  **CheckReadFiltered**$(X, k, bv, c\_r)$;
**endif**
**EndCheckRead**
```

```
**CheckWrite**$(X, k, bv, c\_w)$
**if** $\neg P(X, k, bv_k) \ \wedge \neg F(X, k, bv_k)$ **then**
  **for** $i := 1$ **upto** $k$ **do**
    $F(X, i, bv_i) :=$ **true**;
  **endfor**
  **CheckWriteFiltered**$(X, k, bv, c\_w)$;
**elseif** $P(X, k, bv_k) \ \wedge \ \neg F(X, k, bv_k)$ **then**
  **for** $i := 1$ **upto** $k$ **do**
    $F(X, i, bv_i) :=$ **true**;
  **endfor**
  **CheckR-writeFiltered**$(X, k, bv, c\_w)$;
**endif**
**EndCheckWrite**
```

Fig. 2. Checking to filter read or write access

and three writes. Among the eight accesses, five are level 2 filtered accesses: three filtered reads $\{r1, r2, r6\}$, one filtered write $\{w10\}$, and one filtered r-write $\{w7\}$. The shaded access names in the POEG distinguish the filtered accesses from the other accesses.

Note that the write access $w7$ is a filtered r-write not only in a level 1 iteration but also in a level 2 iteration. It is necessary to record the fact that some accesses, such as $w7$, are filtered accesses in iterations at multiple levels, because these accesses can be involved in races at different levels.

Definition 5. *A* nested filtered access *in a level i iteration is a filtered access, a_j, in a level k iteration $(i < k)$ such that a_j is also a level i filtered access.*

In Figure 1, the filtered write $w10$ in a level 2 iteration is a nested filtered r-write in a level 1 iteration. On the other hand, the filtered read, $r6$, in a level 2 iteration is not a nested filtered access in a level 1 iteration.

In summary of the accesses in Figure 1, we have fifteen accesses to a shared variable X, which include eight filtered accesses, of which five are involved in three first races. The following theorem shows that monitoring the filtered accesses is a sufficient condition for detecting first races in an execution instance. The proof can be found elsewhere [2].

Theorem 1. *If an access a_i is involved in a first race, a_i is a filtered access.*

Determining if each access is a filtered access is solely a function of comparing the current access with other previous accesses to the same shared variable in the iterations at multiple nesting levels. For this purpose, we define two states of nested iteration for each shared variable, which are checked in every access to determine if it is a filtered access.

Definition 6. *A* partially-filtered *iteration for a shared variable X in the i-th nesting level is an iteration in which there exists a level i filtered read to X. A* fully-filtered *iteration for a shared variable X in the i-th nesting level is an iteration in which there exists a level i filtered write or a level i filtered r-write to X.*

Benchmarks		Static		Dynamic			
Kernel	Input Set	Accesses		Access Checks		Filtered Checks	
		Read	Write	Read	Write	Read	Write
MG	$64 \times 64 \times 64$	39	23	347,223,802	25,971,050	68,496,214	19,127,166
FT	$128 \times 128 \times 32$	16	3	221,448	101,380	113,924	101,380
EP	67108864	4	3	93,463,643	26,485,851	41,950,826	26,483,873

Table 1. Race Instrumentation Statistics

Figure 2 shows two algorithms used to check, in each barrier partition bv, if the current read (c_r) or write access (c_w) to a shared variable X is a level k filtered access. The barrier vector, bv, selects a barrier partition for each nesting level. Each thread and barrier operation determines the value of bv for the corresponding partition and nesting level. $P(X, i, bv_i)$ and $F(X, i, bv_i)$ indicate if a level i iteration I_i is partially-filtered and fully-filtered, respectively, in a partition bv_i in I_i for shared variable X. If there are B non-nested barriers in I_i, then $0 \leq bv_i \leq B$, resulting in $(B+1)$ unique variable pairs for I_i. These *private boolean* variables are initialized to *false* at the start of every barrier partition of level i iteration.

Now, look at the procedure **CheckRead()** (**CheckWrite()**). If the condition in line 1 is true, it sets all the $P(X, i, bv_i)$ ($F(X, i, bv_i)$) to true, where $(i \leq k)$ [line 2-4]. If the current read (write) access is the only access in a barrier partition of a level k iteration, it is a filtered read (write) in the iteration and all the level i iterations are guarranteed to be partially-filtered (fully-filtered). And then it performs **CheckReadFiltered()** (**CheckWriteFiltered()**) to invoke the detection protocol incurring the expensive centralized check or serialization. The modifications of $P(X, i, bv_i)$ and $F(X, i, bv_i)$ in the higher level iterations, do not need to be serialized because the value can only change from false to true.

If the condition in line 1 of **CheckRead()** is false, it does nothing, because if an iteration is fully-filtered or partially-filtered in a barrier partition, there exists at least one previous filtered access in the iteration.

In the case of **CheckWrite()**, it tests another condition. If the line 6 condition is true, **CheckWrite()** sets all the $F(X, i, bv_i)$ to true, where $(i \leq k)$ [line 7-9]. If there exists a write access in a barrier partition of a level k iteration, all the level i iterations are fully-filtered. **CheckWrite()** then performs **CheckR-writeFiltered()** to invoke the detection protocol incurring the expensive centralized operation. Otherwise, it does nothing, because if an iteration is fully-filtered in a barrier partition there exists at least one previous filtered write or filtered r-write in the iteration.

Some experiments were performed on a set of three serial NAS benchmarks, in which a small set of common shared variables were monitored and filtered. Table 1 shows that filtering reduced the number of expensive checks to less than half in the case of read accesses, although the benchmarks are fine-grained. The following theorem shows that these algorithms reduce the number of required checks significantly in a monitored execution. The proof appears elsewhere [2].

Theorem 2. *If there exists a set of accesses to a shared variable in an iteration, the set involves at most* $2(B + 1)$ *non-nested filtered accesses in an iteration, where* B *is the number of barrier operations in the iteration.*

4 Related Work

Many approaches for efficiently detecting races on-the-fly for parallel programs have been reported. In this section, we briefly mention some important work to improve the scalability of on-the-fly race detection. This work falls into two groups: compiler support [7] to reduce the number of monitored accesses, and underlying system support [8, 12, 13] using scalable distributed shared memory systems. Our technique is novel in that the scalability is provided with simple but powerful instrumented code which can be applied to most existing techniques.

Mellor-Crummey [7] describes an instrumentation tool for on-the-fly race detection which applies compile-time analysis to identify variable references that need not be monitored at run-time. Using dependence analysis and interprocedural analysis of scalar side effects, the tool was able to reduce the dynamic counts of instrumented operations by 70-100% for the programs tested. Even with the impressive reductions in dynamic counts of monitoring operations, Mellor-Crummey reports that monitoring overhead for run-time detection of data races ran as high as a factor of 5.8.

Min and Choi [8] propose a technique of on-the-fly race detection to minimize the number of times that the monitored program is interrupted for run-time checking of accesses to shared variables. This scheme uses information from the underlying hardware-based, distributed shared-memory, cache coherence protocol and then requires additional hardware support, processor scheduling, cache management and compiler support.

Richards and Larus [13] propose a similar technique to that of Min and Choi in a software-based coherence protocol for a fine-grained data-maintaining distributed shared memory system. To detect data races on-the-fly in programs with barrier-only synchronization, this technique resets access histories at barriers, and monitors only the first read and write after obtaining a copy of a coherence block. They obtain substantial performance improvment but risk missing races. They report an implementation of this technique running on a 32-processor CM-5, and some experiments in which monitored applications had slowdowns ranging from 0-3.

Perković and Keleher [12] implemented on-the-fly race detection in a page-based release-consistent distributed shared memory system, which maintains ordering information that enables the system to make a constant-time determination of whether two accesses are concurrent without compiler support. They extended the system to collect information about the referenced locations and check at barriers for concurrent accesses to shared locations. Although they statically eliminate over 99% of non-shared accesses in applications, an average of 68% of the total overhead in race detection is the run-time overhead to determine whether an access is to shared memory. Nonetheless, the majority of the results are for non-shared accesses. They report that the applications slow down by an average factor of approximately 2.

5 Conclusion

In this paper, we present a new scalable on-the-fly technique for detecting races in parallel programs which may have nested parallelism with barrier synchronization. Our technique reduces the monitoring overhead to require serializing *at most* $2(B+1)$ *non-nested accesses*, *in an iteration* for a shared variable, where B is the number of barrier operations in the iteration. It is important to detect races efficiently, because detecting races might require several iterations of monitoring, and the cost of monitoring a particular execution is still expensive. The technique in this paper can be applied to most existing techniques, therefore, making on-the-fly race detection scalable and more practical for debugging shared-memory parallel programs. We have experimented the technique on a prototype system of race debugging, called *RaceStand* [4], and have been extending it for the programs which have more general types of inter-thread coordination than barrier synchronization.

References

1. Dinning, A., and E. Schonberg, "*An Empirical Comparision of Monitoring Algorithms for Access Anomaly Detection*," 2nd Symp. on Principles and Practice of Parallel Programming, pp. 1-10, ACM, March 1990.
2. Jun, Y., "*Improving Scalablility of On-the-fly Detection for Nested Parallelism*," TR OS-9905, Dept. of Computer Science, Gyeongsang National Univ., March 1999.
3. Jun, Y., and C. E. McDowell, "*On-the-fly Detection of the First Races in Programs with Nested Parallelism*," 2nd Int. Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 1549-1560, CSREA, August 1996.
4. Kim, D., and Y. Jun, "*An Effective Tool for Debugging Races in Parallel Programs*," 3rd Int. Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 117-126, CSREA, July 1997.
5. Kim, J., and Y. Jun, "*Scalable On-the-fly Detection of the First Races in Parallel Programs*," 12th Int. Conf. on Supercomputing, pp. 345-352, ACM, July 1998.
6. Lamport, L., "*Time, Clocks, amd the Ordering of Events in Distributed System*," Communications of ACM, 21(7): 558-565, ACM, July 1978.
7. Mellor-Crummey, J., "*Compile-time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs*," 3rd Workshop on Parallel and Distributed Debugging, pp. 129-139, ACM, May 1993.
8. Min, S. L., and J. D. Choi, "*An Efficient Cache-based Access Anomaly Detection Scheme*," 4th Int. Conf. on Architectural Support for Programming Language and Operating Systems, pp. 235-244, ACM, April 1991.
9. Netzer, R. H., and B. P. Miller, "*Improving the Accuracy of Data Race Detection*," 3rd Symp. on Prin. and Practice of Parallel Prog., pp. 133-144, ACM, April 1991.
10. OpenMP Architecture Review Board, *OpenMP Fortran Application Program Interface*, Version 1.0, Oct. 1997.
11. Parallel Computing Forum, "*PCF Parallel Fortran Extensions*," Fortran Forum, 10(3), ACM, Sept. 1991.
12. Perković, D., and P. Keleher, "*Online Data-Race Detection vis Coherency Guarantees*," 2nd Usenix Symp. on Operating Systems Design and Implementation, pp. 47-58, ACM/IEEE, Oct. 1996.
13. Richards, B., and J. R. Larus, "*Protocol-Based Data-Race Detection*," 2nd Sigmetrics Symp. on Parallel and Dist. Tools, pp. 40-47, ACM, August 1998.