International Journal of Database Theory and Application
Vol.10, No.9 (2017), pp.23-38
http://dx.doi.org/10.14257/ijdta.2017.10.9.03

Efficient Filtering Technique for Reducing Time Overhead of Dynamic Data Race Detection in Multithread Programs*

Ok-Kyoon Ha[1], Se-Won Park[2] and Sung-Phil Heo[3*]

[1]*Department of Aeronautics & Software Engineering, Kyungwoon University*
[2]*Heavy Industries CI Group, Samsung SDS*
[3]*Dept. of Unmanned and Autonomous Vehicle Engineering, Kyungwoon University*
okha@ikw.ac.kr, s-w.park@samsung.com, sungphil.heo@ikw.ac.kr

Abstract

Data races are the hardest defect to handle in multithread programs because they may lead to unpredictable results of the program caused by nondeterministic interleaving of concurrent threads. The main drawback of dynamic data race detection is the heavy additional overhead to monitor and analyze memory operations and thread operations during an execution of the program. It is important to reduce the additional overheads for debugging the concurrency bug. This paper presents a monitoring filtering technique that rules out repeatedly executing regions of parallel loops from the monitoring targets.

***Keywords**: Multithread programs, dynamic data race detection, runtime overheads, monitoring filtering*

1. Introduction

There is an increasingly necessary to write multithread programs due to fact that multi-core and multi-processor systems are commonly used. However, the interleaving of parallel threads may result in concurrency bugs, which are hard to reproduce. Data races in multithread programs are a well-known concurrency defect that they occur when two or more threads access to a shared memory location without explicit synchronizations, and at least one of the accesses is write [1-3]. A multithread program may not exhibit the same execution instance on different runs with the same input. It is difficult to figure out whether a program runs into data races, because there are many possible executions of the program. Detecting data races is therefore important because they may lead to unpredictable results from an execution of the program.

The techniques detecting data races are divided into two broad categories: static analysis and dynamic analysis. The static analysis [4-6] analyzes the defects to use only source codes without any execution. The static analysis is sound, but imprecise because it produces a lot of false positives through evaluation all of possible executions including impractical execution paths which are never reached in the actual execution of the programs. The dynamic analysis employ trace based post-mortem techniques or on-the-fly techniques, which report data races occurred in an execution of a program [7]. The main drawback of dynamic analysis is the additional overhead of monitoring program execution and analyzing every conflicting memory operation. The prior work tries to

Received (December 5, 2016), Review Result (February 16, 2017), Accepted (February 25, 2017)

* This paper is a revised and expanded version of a paper entitled “Empirical Comparison of Filtering Techniques for On-the-fly Data Race Detection in OpenMP Programs” presented at ASEA 2016 conference, November 25, 2016, Jeju, Korea.
* Corresponding Author

ISSN: 2005-4270 IJDTA
Copyright ⓒ 2017 SERSC Australia

reduce the additional runtime overheads, which require from 10 to 100 times than original run. However, there is still room for reducing runtime overheads.

This paper presents a loop filtering technique that rules out repeatedly execution regions of loops from the monitoring targets to provide a minimum runtime overhead. We evaluated the filtering technique under a dynamic data race detection technique, FastTrack [8] and RaceChaser [9], with multithread programs using a huge amount of loop. The empirical results using multithread programs show that the filtering technique reduces the average runtime overhead to 40% of that of dynamic data race detection.

2. Background

Dynamic analysis for detecting data races is precise or imprecise, but unsound since they cannot guarantee to locate the existence of at least one data race in a given execution of the program if there exists any. Dynamic analysis employ trace based post-mortem methods or on-the-fly. On-the-fly methods are based on three different analysis methods: lockset analysis [10-11], happens-before analysis [8,9,12,13], and hybrid analysis[14-15]. The lockset analysis is simple and can be implemented with low overhead. However, lockset analysis may lead to many false positives, because it ignores synchronization primitives which are non-common lock such as signal/wait, fork/join, and barriers. The happens-before analysis is precise, since it does not report false positives and can be applied to all synchronization primitives [7]. However, it is quite difficult to be efficiently implemented due to the performance overheads. The hybrid method tries to reduce the main drawback of pure lockset analysis and to get more improved performance than pure happens-before analysis.

2.1. Redundant Event Filtering

The monitoring filtering techniques are introduced to optimize the performance of on-the-fly data race detection. The filtering techniques exclude unnecessary monitoring of memory operations, such as read only operations and local variables, to reduce the dramatic overheads of the dynamic analysis and to insert minimal monitoring cods. In our prior work [16], we presented a filtering technique, called Redundant Event Filtering (REF), which ignores repeated accesses to a shared memory location. Considering the first access on a thread segment is well-known that it is important to debug a parallel program, because data races involving the first accesses of each thread segment may affect later accesses and may lead to other newly appeared data races. REF basis on an idea to ignore repeated accesses to a shared memory location on a thread segment during monitoring operation.

The data race detection technique of [16] can be firmly established by REF. The dynamic data race detection with REF considers only the first access of each access event type (read and write) in a thread segment, if the accesses to a shared memory location are performed with redundant locks. Given an event e_i, a later same type event e_j than e_i is filtered out by the following conditions:

$$IsFiltered(e_i, e_j) = \begin{cases} \mathcal{T}(e_i) = \mathcal{T}(e_j) \ \wedge \ t_i = t_j \ \wedge \\ t_i.lockset \subset t_j.lockset \ \vee \\ t_i.lockset = t_j.lockset = \emptyset \end{cases}$$

where $\mathcal{T}(e_i)$ represents the event type of e_i, and $t_i.lockset$ is a set of locks living on a thread segment t_i when e_i occurred. Considering the first event on a thread segment is important to debug a multithread program, because data races involving the first events of each thread segment may affect later events and may lead to other newly appeared data races.

 Copyright ⓒ 2017 SERSC Australia

2.2. Memory Hierarchical Filtering

We also presented a hierarchical filtering technique (HIF) [17] that removes non-necessary monitoring operations at the each of three levels, Image Level (IML), Section Level (SEL), and Instruction Level (INL), to reduce runtime overhead of dynamic data race detection.

The objective images which loaded into memory include target binaries for user application, system libraries, and other APIs. However, detecting data race focuses only on the user applications and its libraries rather than system libraries and others. Therefore, HIF selectively filters out memory operations which related standard libraries and other APIs by using the path of linked images and the name of libraries. There exist several dynamic section areas in Data Section, such as global variable area, static variable area, and heap area, that the areas are used to read or write data during the execution of the program. Moreover, the special areas are located for read only data and constant variables. Therefore, HIF filters out such the read only memory areas for SEL. The memory operations should be monitored with each instruction to precisely insert corresponding monitoring codes into the Code Section. The filtering technique analyzes the *opcode* and *operand* of instructions to decide which operations access shared memory locations.

For example, we can estimate two memory locations, *ebp* and *esp* in Figure 1, which are the pointer to stack area of memory by analyzing operand of the instructions and comparing their address with information of stack area. Thus, the almost of memory operations are excluded from the conflicting accesses to memory locations. With HIF, we filter out non-necessary memory operations for dynamic data race detection. Therefore, we are possible to insert monitoring codes into the target binaries with considering only a shared variable *SV*.

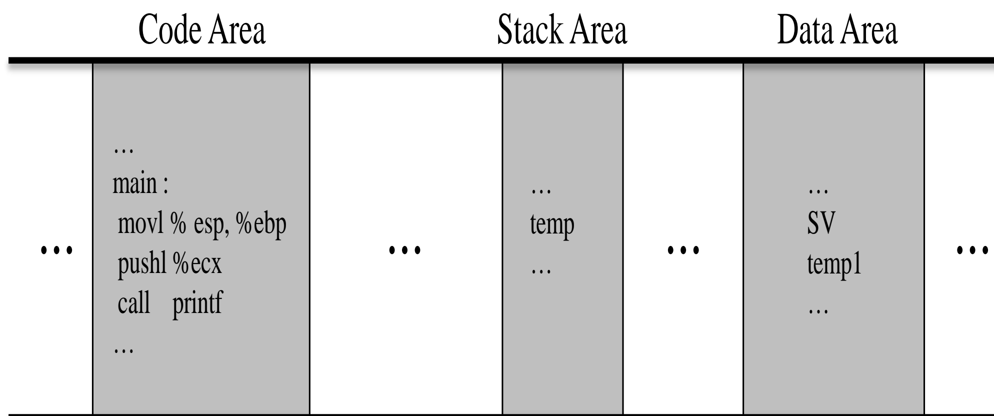

Figure 1. Memory Mapping for an Execution of a Program

With these filtering techniques, we can reduce the average runtime overhead to over 50% of on-the-fly data race detection. However, each of them provides different performance by the characteristic of program executions, such as the number of shared variables and the scale of parallel loops. Thus, we need to compare the effectiveness of the techniques for efficient on-the-fly analysis of parallel programs.

For more optimization of data race detection, we found that there is still room for reducing runtime overhead of dynamic monitoring of multithread programs. When monitoring multithread programs which use a loop-level parallelism with a large number of threads, the runtime overheads depend on the maximum iteration of the loop parallelism. Thus, we address the loop parallelism to reduce the runtime overhead of dynamic data race detection.

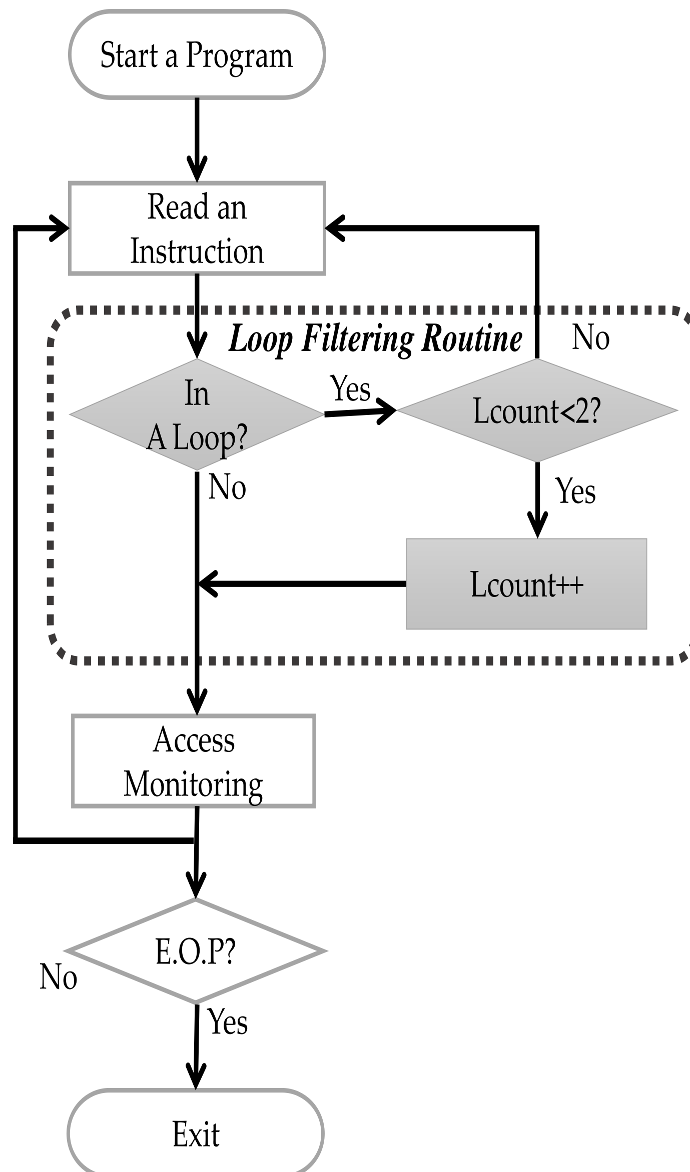

Figure 2. The Process of Monitoring Instructions with Loop Filtering Technique

3. Loop Region Filtering Technique

This section introduces the key idea of Loop Region Filtering (LRF) technique, and presents the design of the monitoring filtering technique including the static area and the dynamic area of LRF.

3.1. The Design of LRF

A loop in a program is intended to repeatedly execute a same code region excluding the usage of special conditions for control the flow of the execution. For detecting data races on-the-fly, it is unnecessary to monitor the same execution of the code regions for every time. Moreover, the monitoring of a same region on a parallel loop with concurrent threads may lead to a same results. Therefore, we can locate data races only by monitoring two threads instead of whole threads for a parallel loop.

Our loop filtering considers only two threads, which are allocated for a parallel loop, to monitor accesses to shared memories. Since it is difficult to identify the common region

 Copyright ⓒ 2017 SERSC Australia

for repeatedly execution by dynamic analysis, we employ a static analysis method to collect the information of the regions. With the static analysis, we collect a LoopId of each loop that consist of a tuple {StartRegion, EndRegion}. Our method dynamically monitors only two threads which occur earlier than others for a parallel loop. Figure 2 shows the process of monitoring instructions including the loop filtering technique. For precisely filtering loop regions, LRF consists of two analysis step: a Static analysis step and a Dynamic analysis step.

3.2. The Design of Static Analysis

The Static analysis step of LRF has two phases: the source code analysis and the object code analysis. In the source code analysis phase aims to identify the start point and end point of loop regions by analyzing the loop statements, such as *for*, *while*, and *do-while*. Figure 3 shows the process of the source code analysis phase.

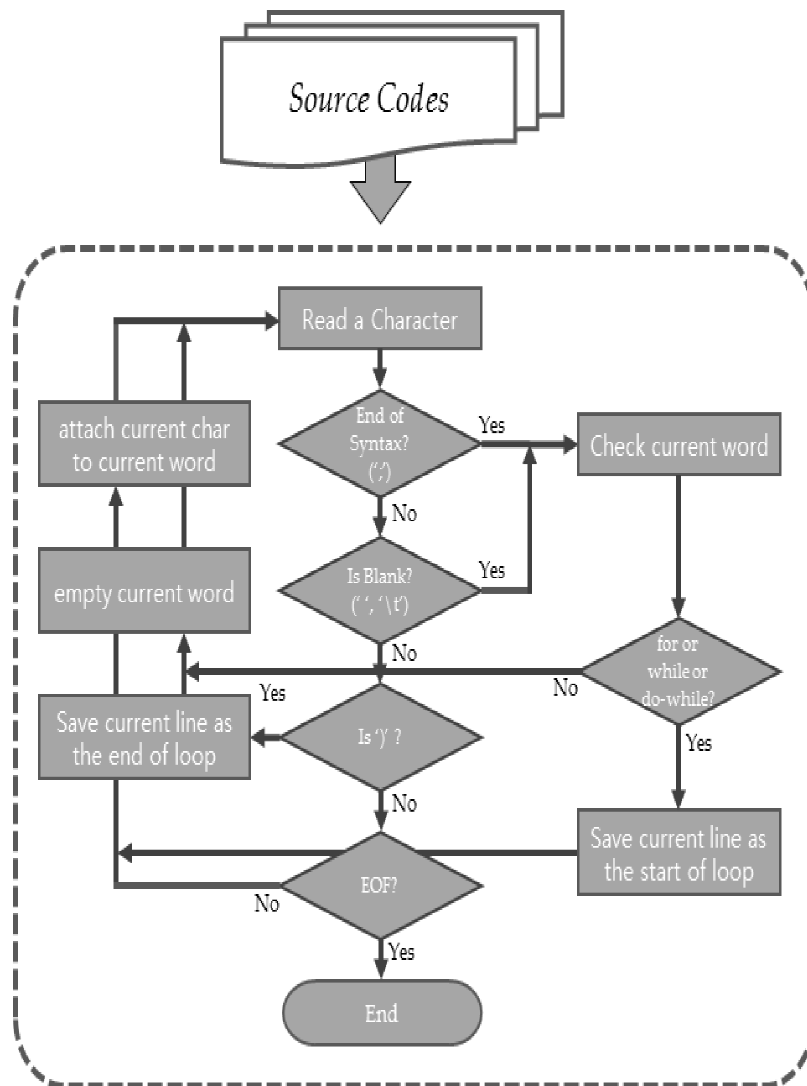

Figure 3. The Source Code Analysis of LRF

The results of the source code analysis consists of a pair of {statement type, the line number of start ~ the line number of end, description}. The statement type is classified into FOR_LOOP, WHILE_LOOP, DO_WHILE_LOOP, and EXPRESSION by the loop statements of the source code. For the object code analysis phase, only the information

about the type of the statements and of the start and end points of the loop statements are needed. However, the details are added to provide information so that the user can review and directly modify the results of the source code analysis. Using these results, it is possible to identify all of loop regions in the source code. Figure 4 shows an example of the result of the source code analysis.

```
FileName : Static_Analysis_Input_Example.c
EXPRESSION<3 ~ 3> : int i,j,x,y
FOR_LOOP<4 ~ 5> : (i=0; i<2; i++)
EXPRESSION<5 ~ 5> : x = 10
FOR_LOOP<6 ~ 6> : (i=0; i<2; i++)
EXPRESSION<6 ~ 6> : x = 15
FOR_LOOP<7 ~ 10> : (i=0; i<2; i++)
EXPRESSION<9 ~ 9> : y = x + i
FOR_LOOP<11 ~ 16> : (i=0; i<2; i++)
EXPRESSION<13 ~ 13> : if(i == 1)
EXPRESSION<14 ~ 14> : break
EXPRESSION<15 ~ 15> : y = x + i
FOR_LOOP<17 ~ 25> : (i=0; i<2; i++)
EXPRESSION<19 ~ 19> : if(i == 0)
EXPRESSION<20 ~ 20> : continue
FOR_LOOP<21 ~ 24> : (j=0; j<2; j++)
EXPRESSION<23 ~ 23> : y = i + j
```

Figure 4. An Example of the Result of the Source Analysis

In the dynamic analysis step, if the loop analysis is compared with the source code analysis for all instructions, the overheads for comparative analysis may occur more than the case where no filtering is applied. Therefore, we design the dynamic analysis to identify the loop regions only by analyzing object code during the static analysis step to identify the shadow memory addresses. We employ a dynamic binary instrumentation (DBI) framework, PIN, for dynamic analysis step. The DBI framework addresses all commands in object code and manages them in shadow memory. Since these shadow memory addresses do not change their addresses during execution unless they are compiled, static addresses can be used in dynamic analysis step as well.

The analysis results of the object code are stored in a file for use in dynamic analysis step and provide a list of identification type, line number, shadow memory address, and the path of the instruction. The identification type includes “new”, “start”, and “end”. “new” means the point immediately before the start of a loop region, and “start” is the point where the repeat is performed first for every iteration within a loop region. “end” is the point immediately after the end of a loop region. The line number and file path are provided to directly remove the non-filtering area using the analysis result. Figure 5 is a result of object code analysis.

 Copyright ⓒ 2017 SERSC Australia

```
new    4  0x400453 Static_Analysis/Static_Analysis_Input_Example.c
start  5  0x400455 Static_Analysis/Static_Analysis_Input_Example.c
end    6  0x400466 Static_Analysis/Static_Analysis_Input_Example.c
new    6  0x40046d Static_Analysis/Static_Analysis_Input_Example.c
start  6  0x40046f Static_Analysis/Static_Analysis_Input_Example.c
end    7  0x400480 Static_Analysis/Static_Analysis_Input_Example.c
new    7  0x400487 Static_Analysis/Static_Analysis_Input_Example.c
start  9  0x400489 Static_Analysis/Static_Analysis_Input_Example.c
end    11 0x40049c Static_Analysis/Static_Analysis_Input_Example.c
new    11 0x4004a3 Static_Analysis/Static_Analysis_Input_Example.c
start  13 0x4004a5 Static_Analysis/Static_Analysis_Input_Example.c
end    17 0x4004be Static_Analysis/Static_Analysis_Input_Example.c
new    17 0x4004c5 Static_Analysis/Static_Analysis_Input_Example.c
start  19 0x4004c7 Static_Analysis/Static_Analysis_Input_Example.c
end    26 0x4004f3 Static_Analysis/Static_Analysis_Input_Example.c
new    21 0x4004d4 Static_Analysis/Static_Analysis_Input_Example.c
start  23 0x4004d6 Static_Analysis/Static_Analysis_Input_Example.c
end    25 0x4004e9 Static_Analysis/Static_Analysis_Input_Example.c
Type      Shadow
          Address
   line                 file path
```

Figure 5. A Result of the Object Code Analysis

3.3. The Design of Dynamic Analysis

Dynamic analysis for data race detection bases on monitoring instructions to check accesses to shared memory locations during an execution of a program. Our loop filtering technique replaces the monitoring process after read an instruction as shown in Figure 2. The filtering technique checks that the current instruction is related in a loop whenever an instruction is read by the LoopId, where a Lcount for each LoopId is employed to count a LoopId execution. Thus, the technique checks the Lcount is less than or equal to 2 to decide continuously monitoring the current instruction. Finally, with our filtering technique, it significantly reduces the runtime overhead of detecting data races.

```
1    LRF_New(tid)
2    {
3        Loop_Stack[tid].Push(0);
4    }

1    LRF_Start(tid)
2    {
3        Loop_Stack[tid].Increas;
4    }

1    LRF_End(tid)
2    {
3        Loop_Stack[tid].Pop( );
4    }
```

Figure 6. Algorithm for Dynamic Analysis of LRF

The key to dynamic analysis is to identify the “new”, “start”, and “end” points of the loop regions. It can be simply filtered out by checking whether it is excluded from monitoring by loop area filtering before monitoring all memories. Figure 6 shows the dynamic analysis algorithms for “new”, “start”, and “end” point of the loop region.

Copyright ⓒ 2017 SERSC Australia

4. Implementation and Experimentation

4.1. Implementation

We implemented LRF into a dynamic data race detector which use PIN dynamic binary instrumentation framework [18-19]. Figure 7 shows the architecture of a dynamic detector with the filtering technique. Our filtering technique run on top of the PIN during detection of data races by Detector. A hash algorithm and a stack structure were used in the loop filtering technique to remove the centralized bottleneck of accessing LoopId. For the implementation, we considered an exception that a thread executes a same loop region for two times because it may lead to miss data races in the region. The implementations were carried on a system with Intel® Xeon 2.4 GHz CPU and 48GB Memory under CentOS using Kernel 2.6. To evaluate the filtering technique, we employed FastTrack and RaceChaser algorithm as a dynamic detector.

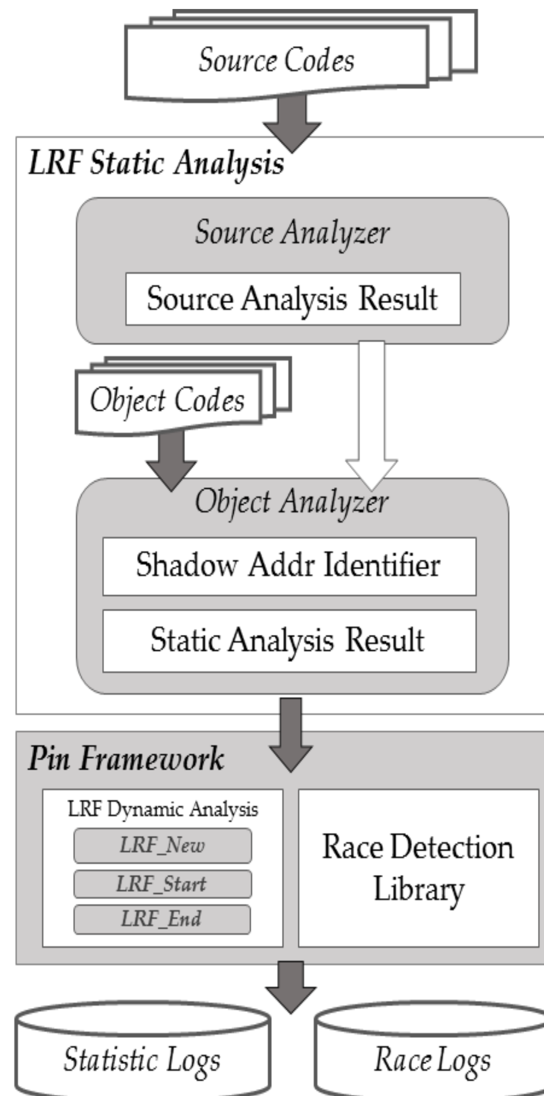

Figure 7. The Overall Architecture of Dynamic Detector with LRF

To combine the data race detector with LRF, the source code of the RaceChaser was partially modified. The modified RaceChaser enables access to shared variables using the "IMG_AddinstrumentFunction" function of PIN and also enables access to thread

 Copyright ⓒ 2017 SERSC Australia

information using the “PIN_AddThreadStartFunction” and “PIN_AddThread FinFunction” functions. The routine of LRF was inserted into the “IMG_AddinstrumentFunction”.

The modified detector obtains the first section address of the object file by calling "IMG_SecHead" function as the parameter of the current binary image. The tool obtains the addresses of the routines by calling "SEC_RtnHead" with the section addresses, and calls "RTN_InsHead" with the routine addresses to collect the shadow addresses of the instructions. The detector repeatedly also calls all the "SEC_Next", "RTN_Next", "INS_Next" to check all of instruction addresses, and it compares the addresses of each instruction with the shadow memory addresses of the static analysis result file. If a matching address is found, it inserts proper function defined by the identification type of the result file using "INS_InsertCall".

4.2. Design of Experimentation

The implementations were carried on a system with Intel® Xeon 2.4 GHz CPU and 48GB Memory under CentOS using Kernel 2.6. To evaluate the filtering technique. We developed the synthetic programs considering two criteria such as using sing loop and using serialized loops. The designed synthesis also consider read-only access and read-write accesses to identify data race detection. The synthetic programs appear in Table 1. We measured the runtime overheads of detecting data races under both pure detector and filtered detector using the loop filtering. For the experimentation, each synthetic program has the maximum number of iterations for a parallel loop which ranges from 50K to 5000K.

Table 1. Design of Synthetic Programs

	Read-Only Accesses	Read-Write Accesses
Single Loop	Single-RO	Single-RW
Serialized Loops	Serial-RO	Serial-RW

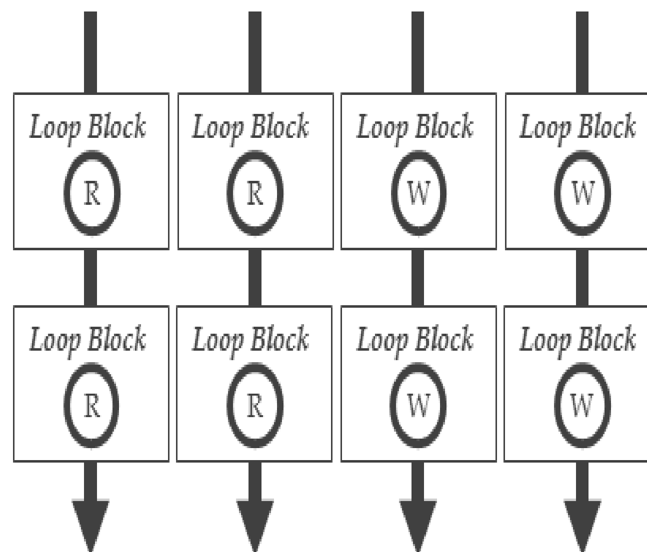

Figure 8. Design of Serial-RW Program

We also empirically compared the efficiency of LRF with REF using OpenMP bechmarks. For the experiments, we consider four cases, Non-Filtering, With-REF, With-LRF, and With-All, and measure the runtime overheads of each case.

Copyright ⓒ 2017 SERSC Australia

We chose four applications, *FFT6*, *MD*, *Mandelbrot*, and *PI*, from the *OmpSCR* (the OpenMP Source Code Repository) benchmark set considering the features of programs, such as the number of shared variables and the scale of parallel loops. Their features are specified in Table 2.

Table 2. The Features of OpenMP Benchmarks

Applications	Lines	Accesses		Locks	Loop Count
		Read	Write		
FFT6	542	2285K	15399K	1	38K
MD	266	23584K	9451K	0	5632
Mandelbrot	144	114537K	8	0	1024
PI	83	20000K	8	0	50000K

4.3. Results and Analysis

We verified the accuracy of the static analysis of LRF using the synthetic programs. Figure 9 shows the source code analysis result for Serial-RW program. From the result, we see that lines 8 to 9, lines 10 to 11, lines 17 to 18, and lines 19 to 20 are loop regions.

```
FileName : Se-RW.c
EXPRESSION<1 ~ 1> : #include <stdio.h>
EXPRESSION<2 ~ 2> : #include <pthread.h>
EXPRESSION<2 ~ 3> : #define N 50000
EXPRESSION<4 ~ 4> : int shared_var
EXPRESSION<5 ~ 5> : void *read_thread(void *arg)
EXPRESSION<6 ~ 6> : {
EXPRESSION<7 ~ 7> : int i, temp
FOR_LOOP<8 ~ 9> : (i=0; i<N; i++)
EXPRESSION<9 ~ 9> : temp = shared_var
FOR_LOOP<10 ~ 11> : (i=0; i<N; i++)
EXPRESSION<11 ~ 11> : temp = shared_var
EXPRESSION<12 ~ 12> : return NULL
EXPRESSION<13 ~ 13> : }
EXPRESSION<14 ~ 14> : void *write_thread(void *arg)
EXPRESSION<15 ~ 15> : {
EXPRESSION<16 ~ 16> : int i
FOR_LOOP<17 ~ 18> : (i=0; i<N; i++)
EXPRESSION<18 ~ 18> : shared_var = i
FOR_LOOP<19 ~ 20> : (i=0; i<N; i++)
EXPRESSION<20 ~ 20> : shared_var = i
EXPRESSION<21 ~ 21> : return NULL
EXPRESSION<22 ~ 22> : }
EXPRESSION<23 ~ 23> : int main()
EXPRESSION<24 ~ 24> : {
EXPRESSION<25 ~ 25> : pthread_t tid[4]
EXPRESSION<26 ~ 26> : pthread_create(&tid[0], NULL, read_thread, NULL)
EXPRESSION<27 ~ 27> : pthread_create(&tid[1], NULL, read_thread, NULL)
EXPRESSION<28 ~ 28> : pthread_create(&tid[2], NULL, read_thread, NULL)
EXPRESSION<29 ~ 29> : pthread_create(&tid[3], NULL, read_thread, NULL)
EXPRESSION<30 ~ 30> : pthread_join(tid[0], NULL)
EXPRESSION<31 ~ 31> : pthread_join(tid[1], NULL)
EXPRESSION<32 ~ 32> : pthread_join(tid[2], NULL)
EXPRESSION<33 ~ 33> : pthread_join(tid[3], NULL)
EXPRESSION<34 ~ 34> : return 0
EXPRESSION<35 ~ 35> : }
```

Figure 9. The Source Code Analysis Result for Serial-RW

 Copyright ⓒ 2017 SERSC Australia

Table 3 presents the results of data race detection by using LRF and non-applied LRF for each synthesis.

Table 3. The Results of Data Race Detection for Synthetic Programs

Detectors	Synthesis	Without LRF	With LRF
FastTrack	Single-RO	0	0
	Serial-RO	0	0
	Single-RW	2	2
	Serial-RW	6	6
RaceChaser	Single-RO	0	0
	Serial-RO	0	0
	Single-RW	2	2
	Serial-RW	6	6

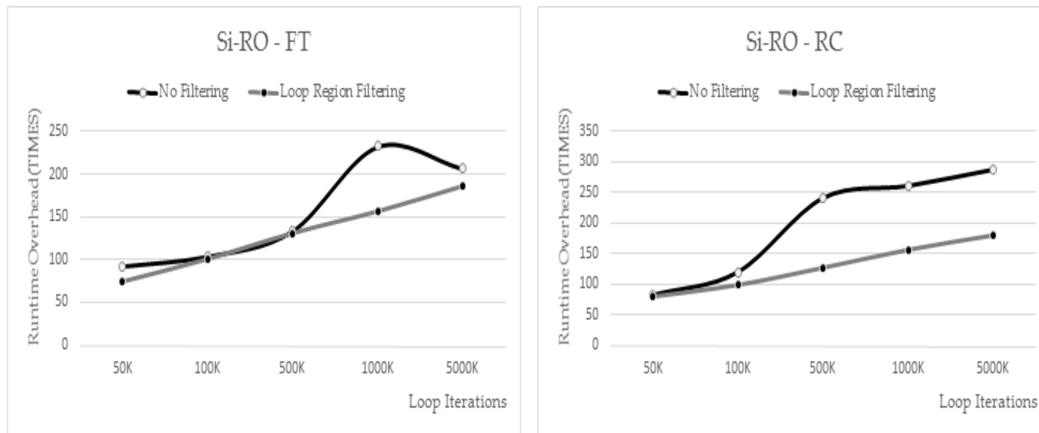

Figure 10. The Results of Runtime Overheads for Single-RO

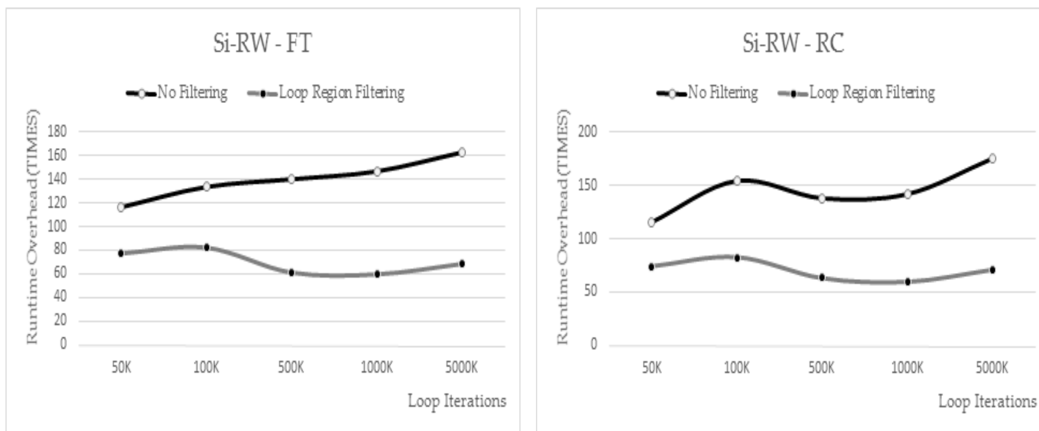

Figure 11. The Results of Runtime Overheads for Single-RW

Copyright ⓒ 2017 SERSC Australia

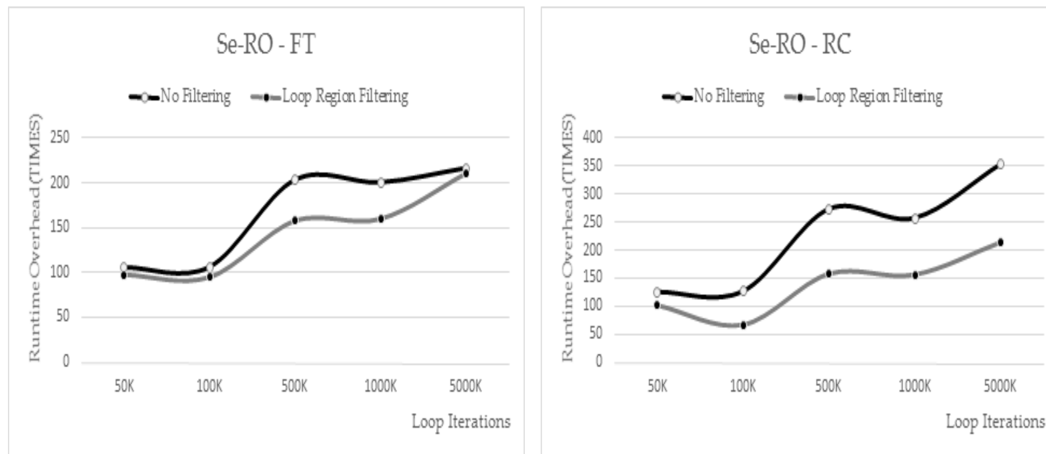

Figure 12. The Results of Runtime Overheads for Serial-RO

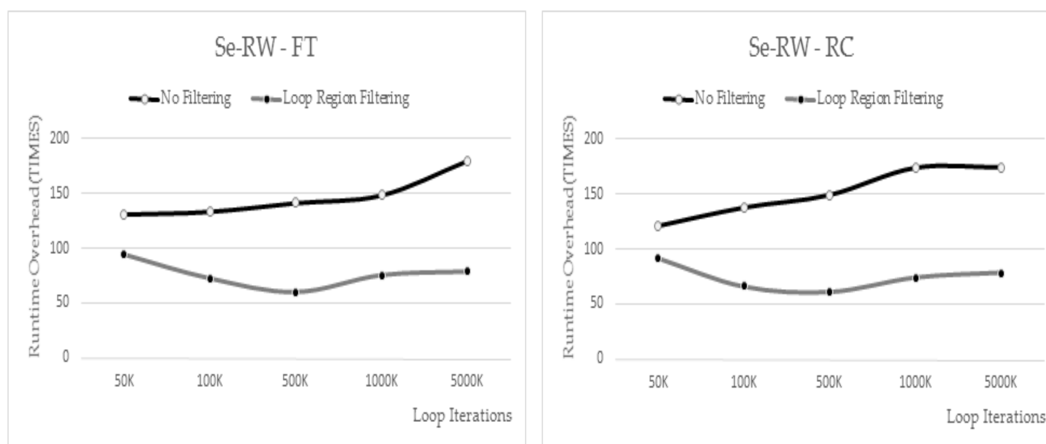

Figure 13. The Results of Runtime Overheads for Serial-RW

From Figure 10 to Figure 11 show the results of measured runtime overheads for single loops programs. In the figures, the detectors with filtering incurred an average runtime overhead of 33.0x, whereas the detectors without filtering incurred an average runtime overhead of 97.5x. In Figure 11, we can see that the runtime overheads are less than the results of Single-RO due to the additional runtime for write accesses. However, the results in Figure 11 were included for LRF. The measured results for serialized loop programs appear in Figure 12 and Figure 13. From the figures, we see that the detector with filtering incurred an average runtime overhead of 30.5x, whereas the detector without filtering averaged more than 104x slowdown.

From the empirical results, our loop filtering technique not only reduces runtime overheads of detecting data races but also provides a fix overhead to monitor accesses to shared memories, while the detector without filtering depends on the maximum iteration of the loop parallelism. Finally, the filtering technique reduces the average runtime overhead to 60% of that of pure data race detection.

 Copyright ⓒ 2017 SERSC Australia

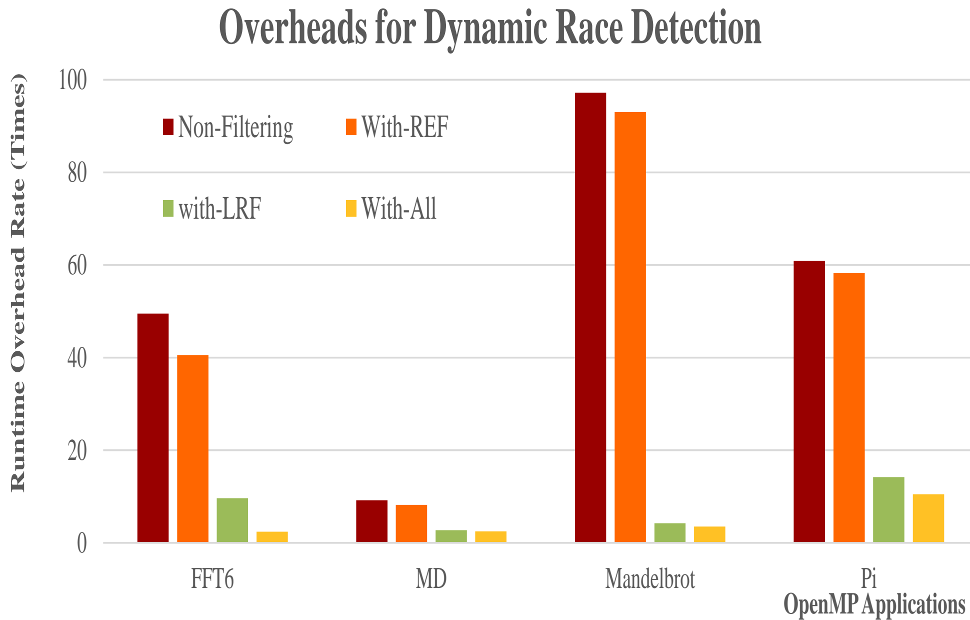

Figure 14. The Measured Results of Runtime Overhead for OpenMP Programs

We measured the runtime of the benchmarks over the four cases of on-the-fly data race detection using RaceChaser. Figure 14 shows the average runtime overhead for each case as a proportion of the original run. In the figure, Non-Filtering means the pure detection without any filtering techniques. With-REF and With-LRF means that we measured the runtime overhead of the execution of benchmarks under detection with each filtering technique, and With-All indicates the measured results that the runtime overhead under dynamic detection with both REF and LRF.

As shown in the results of Figure 14, the With-REF case and the With-LRF case reduces the average runtime overhead to 92.2% and 14.2%, respectively, of that of Non-Filtering case. Moreover, the dynamic data race detection incurred only an average runtime overhead of 8.7% than the Non-Filtering case. The empirical results show that the With-All case is practical method for on-the-fly data race detection.

5. Conclusion

It is important to reduce the additional overheads for dynamic detection of data races in multithread programs. This paper presents a loop filtering technique that rules out repeatedly execution regions of loops from the monitoring targets in the programs. The loop filtering technique not only reduces runtime overheads of detecting data races but also provides a fix overhead to monitor accesses to shared memories, while the detector without filtering depends on the maximum iteration of the loop parallelism. We compared the runtime overheads of detecting data races under both pure detector and filtered detector using the loop filtering. The empirical results using multithread programs show that the filtering technique reduces the average runtime overhead to 60% of that of pure data race detection. We also empirically compared the efficiency of two monitoring filtering techniques, REF and LRF, which reduces the dramatic overheads of the dynamic analysis by excluding unnecessary monitoring memory operations. The experimental results using OpenMP benchmarks show that the case of the detection with both REF and LRF is practical for on-the-fly data race detection, since it reduces the average runtime overhead to under 10% of that of the pure detection.

Copyright ⓒ 2017 SERSC Australia

Acknowledgments

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2060082), and also was supported by a grant (#S0144-15-1007) from Gyungbuk Software Convergence Cluster Project funded by MSIP(Ministry of Science, ICT and Future Planning) and NIPA(National IT Industry Promotion Agency).

References

[1] R. H. B. Netzer and B. P. Miller, "What are Race Conditions?: Some Issues and Formalizations", in Proceeding of Letters on Programming Languages and Systems, LOPLAS 1992, ACM, **(1992)**, pp. 74-88.

[2] E. Farchi, Y. Nir and S. Ur, "Concurrent Bug Patterns and How to Test Them", Proceeding of the 17th International Symposium on Parallel and Distributed Processing, IPDPS 2003. IEEE, **(2003)**, pp. 7.

[3] U. Banerjee, B. Bliss, Z. Ma and P. Petersen, "A Theory of Data Race Detection", Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging, PADTAD 2006, ACM, pp. 69-78, **(2006)**.

[4] D. Callahan and J. Sublok, "Static Analysis of Low-level Synchronization", SIGPLAN Not., ACM, vol. 24, **(1988)**, pp. 100-111.

[5] C. E. McDowell, "A Practical Algorithm for Static Analysis of Parallel Programs", Journal of Parallel and Distributed Computing, Springer-verlag, vol. 6, no. 3, **(1989)**, pp. 515-536.

[6] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks", Proceedings of the nineteenth ACM symposium on Operating systems principles, SOSP 2003, ACM, **(2003)**, pp. 237-252.

[7] O.-K. Ha, "Case Study of Dynamic Detectors for Data Races", In Proceedings of International Conference on Electronic Engineering and Computer Science (EECS 2013), IERI Procedia, Beijing, China, vol. 4, **(2013)**, pp. 174-180.

[8] C. Flanagan and S. N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection", Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, ACM, **(2009)**, pp. 121-133.

[9] O.-K. Ha and Y.-K. Jun, "An Efficient Algorithm for On-the-Fly Data Race Detection Using an Epoch-Based Technique", In Journal of Scientific Programming, Article No. 13, **(2015)** January.

[10] 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), ACM, vol. 15, **(1997)**, pp. 391-411.

[11] H. Nishiyama, "Detecting Data Races using Dynamic Escape Analysis based on Read Barrier", In Proceedings of the 3rd conference on Virtual Machine Research and Technology Symposium, Berkeley, CA, USA, 2004. USENIX Association, vol. 3, pp. 10-10.

[12] E. Pozniansky and A. Schuster, "Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs", Proceeding of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP 2003, ACM, **(2003)**, pp. 179-190.

[13] A. Jannesari and W. F. Tichy, "On-the-fly Race Detection in Multi-threaded Programs", Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging, PADTAD 2008, ACM, **(2008)**, pp. 6:1-6:10.

[14] T. Elmas, S. Qadeer and S. Tasiran, "Goldilocks: A Race and Transaction-aware Java Runtime", Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI 2007, ACM, **(2007)**, pp. 245-255.

[15] A. Jannesari, B. Kaibin, V. Pankratius and W. F. Tichy, "Helgrind+: An Efficient Dynamic Race Detector", Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, IPDPS 2009, IEEE Computer Society, **(2009)**, pp. 1-13.

[16] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue and Y.-K. Jun, "On-the-fly Detection of Data Races in OpenMP Programs", In Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD'2012), ACM, Minneapolis, USA, **(2012)** July, pp. 1-10.

[17] O.-K. Ha and Y.-K. Jun, "Effective Monitoring Memory Operations for Dynamic Race Detection through Hierarchical Filtering Method", In International Journal of Multimedia and Ubiquitous Engineering, **(2014)**, pp. 199-208.

[18] H. Patil, C. Pereira, M. Stallcup, G. Lueck and Cownie, "Pinplay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs", Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, CGO 2010, ACM, **(2010)**, pp. 2-11.

[19] A. R. Bernat and B. P. Miller, "Anywhere, Any-time Binary Instrumentation", Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, PASTE 2011, ACM, **(2011)**, pp. 9-16.

Copyright ⓒ 2017 SERSC Australia

[20] S.-W. Park, O.-K. Ha and Y.-K. Jun, "A Loop Filtering Technique for Reducing Time Overhead of Dynamic Data Race Detection", In Proceedings of the 8th International Conference on Database Theory and Application (DTA 2015), IEEE, Jeju, Korea, **(2015)**, pp. 29-32.

Authors

Ok-Kyoon Ha received the BS degree in Computer Science under the Bachelor's Degree Examination Law for Self-Education from National Institute for Lifelong Education, and the MS and PhD degree in Informatics from Gyeongsang National University (GNU), South Korea. He is now a professor of department of aeronautics & software engineering in Kyungwoo University, South Korea. He worked as the manager of IT department in Korea industry for several years. His research interests including parallel/distributed programming, software testing and debugging, embedded system programs, dependable software, and software development activities for avionics. Dr. Ha is a member of Korean Institute of Information Technology (KIIT), Korea Institute of Information Scientist and Engineers (KIISE), and Korea Society of Computer Information (KSCI).

Se-Won Park received the BS degree in Informatics, and the MS degree in Informatics from Gyeongsang National University (GNU), South Korea. He is now a software engineer for Heavy Industries CI Group in Samsung SDS, South Korea. His research interests including parallel/distributed programming and its debugging, embedded system programs, and dependable software. Mr. Park is a member of Korea Institute of Information Scientist and Engineers (KIISE) and Korea Society of Computer Information (KSCI).

Sung-Phil Heo received the Ph.D. degree in Information Sciences from Tohoku University, Sendai, Japan, in 2004. From 1993 to 2014, he was a Principle Researcher, Team Leader, and Project Director in the Korea Telecom R&D Center, Seoul, Korea. He joined Kumoh National Institute of Technology in 2014, where he was a professor of ICT Convergence Research Center. Currently he is an associate professor of Dept. of Unmanned and Autonomous Vehicle Engineering, Kyungwoon University, South Korea. His research interests include IoT/M2M, contents based multimedia retrieval, and next generation wireless communication technology.

Copyright ⓒ 2017 SERSC Australia

 Copyright ⓒ 2017 SERSC Australia