Empirical Comparison of Filtering Techniques for On-the-fly Data Race Detection in OpenMP Programs

Ok-Kyoon Ha¹ and Sung-Phil Heo²

¹Department of Aeronautics & Software Engineering, Kyungwoon University,
²Department of Unmanned and Autonomous Vehicle Engineering, Kyungwoon University,
730, Gangdongro, Gumi, Gyeongbuk, 39160, Republic of Korea
{okha, sungphil.heo}@ikw.ac.kr

Abstract. It is a well-known that data races in implicit threading applications, such as OpenMP programs, are the most notorious class of concurrency bugs, because they may lead to unpredictable results of the program. The main drawback of on-the-fly data race detection techniques is the heavy additional overheads for analyzing every memory operations and thread operations, such as load, store, fork, and join. It is important to reduce the additional overheads for debugging the concurrency bug. This paper presents an empirical comparison of two filtering techniques which are applied to parallel programs with OpenMP directives for dynamic data race detection.

Keywords: OpenMP programs, dynamic data race detection, monitoring filtering techniques.

1 Introduction

OpenMP [1] is the de facto standard for shared memory programs supporting a nestable fork-join model of parallel execution. It provides a portable way to achieve high performance with features like nested parallelism and simple compiler directives to transform a sequential program into parallel program [2]. However, the interleaving of parallel threads may result in concurrency bugs, such as deadlocks and data races, which are hard to reproduce. Data races in OpenMP 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 [3]. A parallel 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, since they may lead to unpredictable results from an execution of the program.

The techniques for detecting data races are divided into two broad categories: static analysis, which analyzes the detect to use only source codes without any execution, and dynamic analysis, which employs reports data races occurred in an execution of the programs [4]. The dynamic analysis employs trace based post-mortem techniques or on-the-fly techniques [5-6]. The main drawback of the dynamic analysis is the
additional overheads to monitor program execution and to analyze every conflicting memory and thread operations [2]. The prior work tries to reduce the additional runtime overheads, which require from 10 to 100 times than original run, during monitoring executions or analyzing operations. However, there is still room for reducing runtime overheads.

2 Filtering Techniques for Dynamic Data Race Detection

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 codes. In prior works, we presented two filtering techniques, Redundant Event Filtering (REF) [2] and Loop Region Filtering (LRF) [7] that can be applied to the implicit threading model such as OpenMP.

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 during monitoring operation. Thus, on-the-fly data race detection with REF considers only the first access of each access type (read and write) in a thread segment, if the accesses to a shared memory location are performed with redundant locks.

LRF basis on the insight that the full monitoring of threads is often unnecessary for detecting data races in parallel programs which use a loop-level parallelism with a large number of threads. The key idea behind this insight is that a loop in the program is intended to repeatedly execute a same code region excluding the usage of control flows, such as conditional statements, and the monitoring of a same region on a parallel loop with concurrent threads may lead to a same output. So that, LRF considers only two threads, which are allocated for a parallel loop, to monitor accesses to shared memories. Therefore, detecting data races with LRF can locate data races only by monitoring two threads instead of whole threads.

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.

3 Experimentation and Results

We implemented a data race detector including two filtering techniques, REF and LRF, on top of Pin binary instrumentation framework which widely used to analyze an execution of a program dynamically, and empirically compare the efficiency of the filtering techniques using OpenMP benchmarks. For the experiments, we consider
four cases, Non-Filtering, With-REF, With-LRF, and With-All, and measure the runtime overheads of each case.

Table 1. The features of OpenMP benchmarks

<table>
<thead>
<tr>
<th>Applications</th>
<th>Lines</th>
<th>Accesses</th>
<th>Locks</th>
<th>Loop Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Read</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>FFT6</td>
<td>542</td>
<td>2285K</td>
<td>15399K</td>
<td>1</td>
</tr>
<tr>
<td>MD</td>
<td>266</td>
<td>23584K</td>
<td>9451K</td>
<td>0</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>144</td>
<td>114537K</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>PI</td>
<td>83</td>
<td>20000K</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

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 1. For the experimentation, we used a system with Intel Xeon Quad-core 2 CPUs and 48GB main memory under the Kernel 2.6 of Linux operating system, and also used gcc 4.1.2 to compile our runtime libraries and the benchmarks.

Table 2. The measured results of runtime overhead

<table>
<thead>
<tr>
<th>Applications</th>
<th>Non-Filtering</th>
<th>With-REF</th>
<th>With-LRF</th>
<th>With-ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT6</td>
<td>49.5</td>
<td>40.5</td>
<td>9.6</td>
<td>2.4</td>
</tr>
<tr>
<td>MD</td>
<td>9.2</td>
<td>8.2</td>
<td>2.7</td>
<td>2.5</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>97.2</td>
<td>93.0</td>
<td>4.2</td>
<td>3.5</td>
</tr>
<tr>
<td>PI</td>
<td>60.9</td>
<td>58.2</td>
<td>14.2</td>
<td>10.5</td>
</tr>
</tbody>
</table>

We measured the runtime of the benchmarks over the four cases of on-the-fly data race detection. Table 2 shows the average runtime overhead for each case as a proportion of the original run. In the table, 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 the table, 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.
4 Conclusion

In this paper, we empirically compared the efficiency of two representative 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.

Acknowledgments. This research 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