Using Escape Analysis in Dynamic Data Race Detection

Williams College Technical Report CSTR201401

Emma Harrington
Williams College

Stephen N. Freund
Williams College

1. INTRODUCTION

Data race conditions occur when multiple threads concurrently access a memory location and at least one access is a write. Race conditions are easy to introduce and hard to detect because they may not always lead to incorrect behavior. Dynamic analysis tools can detect data races automatically, but their use comes at a performance cost since precise detectors must check every access to every variable.

In many multithreaded programs, this cost can be mitigated by leveraging the observation that some objects are only accessible from their creating thread and thus cannot be involved in a race. We show that by tracking whether objects are thread-local, dynamic race detectors can skip many checks and thereby improve performance, especially when a static analysis identifies some thread-local objects at compile time.

2. BACKGROUND

Dynamic race detectors find data races. Anyone who has shared files with a coworker is familiar with these bugs. Imagine that one day you and I are working on the same file. While you work on the old version, I update the file. Unaware of the changes, you update the file again and overwrite my work. Every time we work on the same file we create a "race condition," where we can inadvertently step on each other's work. Frustrated by our lost work and fearing that other losses have gone unnoticed, we decide to protect each file with a mutual exclusion lock. Using the terminology of concurrent programming, if I acquire a lock and access the corresponding file, you must wait until I finish and release the lock before you access the file.

In concurrent software, the danger of conflicting accesses comes from multiple threads, each of which is executed like a standalone program with its own instructions and control structure but with access to data shared by all the threads. To avoid races, programmers writing concurrent programs use a locking discipline like that above to coordinate their program's threads.

While locks can help avoid errors, it is all too easy for a programmer to forget to acquire a necessary lock or to acquire the wrong one, allowing one thread to overwrite another's work in hard to detect ways. Fortunately, programmers can use dynamic race detectors to alert them when data is not properly protected. These detectors, however, degrade performance because they must track every access to every variable and verify that the access is race free.

In our research, we explore ways to speed up dynamic race detectors by eliminating checks on accesses to thread-local objects, which are guaranteed to be free from races. An object is thread-local if it is only accessible from its creating thread because references to it have not "escaped" to other threads. Like files only available to a single author, thread-local objects cannot be involved in races and can be ignored by race detectors.

Many studies focus on statically identifying unnecessary checks like those on thread-local objects [6, 3]. Unfortunately, static escape analyses are inherently imprecise due to the fundamental limitation of what computers can compute. Static analyses, therefore, conservatively label fewer objects thread-local than will be thread-local at runtime [7]. Our approach is unique because we implement a dynamic escape analysis to precisely eliminate all checks on thread-local objects. We then develop a hybrid static and dynamic escape analysis that aims to achieve the best of both worlds, by continuing to filter out all thread-local accesses while saving work at runtime by identifying objects that are thread-local on every program execution and hence can always be ignored safely.

3. APPROACH

Our research questions are:

1. How common are local accesses?
2. Does eliminating checks on local accesses speed up race detection?
3. Does a hybrid static and dynamic escape analysis really get the best of both worlds?

To answer the first question, we extended the RoadRunner dynamic analysis framework for Java [5] with an escape analysis to identify which objects are accessible from multiple threads. We then subtracted the number of accesses made on these shared objects from the total number of accesses in the program to determine how many accesses are made to thread-local objects. These thread-local accesses represent checks that a dynamic escape analysis could filter out for a precise race detector. Hence, the number of these accesses serves as a good approximation of the potential of our approach.

The second question is whether stripping checks on accesses to thread-local objects can speed up race detection. To be

faster, the time spent by the dynamic escape analysis to determine whether accesses to objects can be safely ignored must be less than that spent by race checks on those accesses. To test this, we added a dynamic escape analysis to the FastTrack dynamic race detector [4]. As described in Section 4, there were some performance gains, but it was clear that dynamic escape analysis was fairly expensive.

To improve on the runtimes of the purely dynamic tool, we investigated moving some of the escape analysis to compile time. Our static filter uses the thread-local analysis in the WALA static analysis framework [2] to determine which classes are always thread-local and hence safely ignored by FastTrack. This filter can be applied directly to FastTrack or in conjunction with our dynamic escape analysis. By using it with the dynamic escape analysis, we can reduce the runtime overhead without sacrificing the precision of the escape analysis.

4. RESULTS

We tested our approach on eleven benchmarks, including the multi-threaded Java Grande Benchmark series [1] and others [6]. We first estimated the potential of our approach by determining the number of objects that remain local to their parent thread in the benchmark programs. As shown in Figure 1, about half of the objects in the benchmarks remain local to their creating thread throughout their lifetime (shown in black).

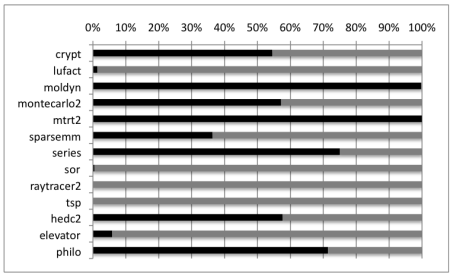

Figure 1: Local vs shared objects.

In the benchmarks, the local vs. shared accesses have a similar breakdown. While shared objects are accessed disproportionately often, about 40% of accesses are local and can be safely ignored by FastTrack. For some of the benchmarks, almost all of the state and accesses are local. This suggests that certain types of multithreaded programs may have mostly local state interspersed with a few shared objects. A dynamic escape analysis could dramatically reduce the overhead of race detection on these quasi-concurrent programs.

We tested dynamic escape analysis as a filter to FastTrack. With dynamic escape analysis, three of the ten benchmarks in the test suite saw speedups of up to 24% compared to the FastTrack tool run in the same configuration. However, other benchmarks ran slower because the analysis to identify thread-local objects took longer than simply checking the accesses for races.

The static escape analysis made our dynamic one more effective: seven out of the ten benchmarks saw speedups after adding the dynamic escape analysis on top of the static filter with gains as high as 33% and 8% on average.

5. CONTRIBUTIONS AND FUTURE WORK

The contributions of our work are:

1. We establish that race detectors spend a significant amount of time checking accesses to thread-local data that are inherently race free.
2. We demonstrate the potential of using dynamic escape analysis to identify objects that are only accessible from their creating thread and then filter out accesses to these thread-local objects from the race detector.
3. We show how a dynamic escape analysis can be augmented with a static one to maintain precision while moving work to compile time.

In future work, we want to further reduce the overhead of tracking the shared status of objects. To identify shared objects, we want to piggyback on the similar activities performed by the garbage collector, which must identify garbage objects reachable from no threads. The interaction between concurrency and garbage collection is discussed in [7].

We also want to experiment with using concurrency to reduce the overhead of the dynamic escape analysis. In a simple test, we harnessed extra processors to perform traversals of newly shared objects, moving this work off the cores running the target program. For our benchmark programs, the overhead associated with spawning new threads, however, trumped the potential gains from added concurrency. Since new threads on the main CPU suffer from high overhead, in future work, we could switch to lighter weight threads of the GPU, which could offer the same boon of extra processing at a lower start-up cost.

6. SUMMARY

With the rise of multi-core computers, concurrent software has become more widespread and tools for detecting their bugs more important. Ideally these tools will be both precise and speedy, reporting all bugs with no false positives and minimal overhead. Tools like FastTrack [4] are precise but costly because they must check every access to every variable in the target program. While some tools trade off precision for speed, a growing body of work focuses on eliminating unnecessary checks. We add to this research by eliminating unnecessary checks through dynamic escape analysis and comparing this approach to a simple static analysis and a hybrid combination of the two.

7. ACKNOWLEDGEMENTS

This work was supported by NSF Grants 1116825 and 1421051.

8. REFERENCES

[1] The Java Grande Multi-threaded Benchmarks.
[2] T.J. Watson Libraries for Analysis (WALA), 2012.
[3] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

[4] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, 2010.

[5] C. Flanagan and S. N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 1–8, 2010.

[6] C. Flanagan and S. N. Freund. Redcard: Redundant check elimination for dynamic race detectors. In *Proceedings of European Conference on Object-Oriented Programming*, pages 255–280, 2013.

[7] T. Kalibera, M. Mole, R. E. Jones, and J. Vitek. A black-box approach to understanding concurrency in dacapo. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 335–354, 2012.