Mößbauer, Felix:

High Performance Dynamic Threading Analysis for Hybrid Applications

Masterarbeit, Wintersemester 2019

Gutachter: Kranzlmüller, Dieter

Fakultät für Mathematik, Informatik und Statistik
Institut für Informatik
Master Informatik

Ludwig-Maximilians-Universität München

https://doi.org/10.5282/ubm/epub.60621

Studienabschlussarbeiten
Fakultät für Mathematik, Informatik
und Statistik

Felix Mößbauer:

High Performance Dynamic Threading Analysis for Hybrid Applications

Masterarbeit, Wintersemester 2019

Gutachter: Dieter Kranzlmüller

Fakultät für Mathematik, Informatik und Statistik
Institut für Informatik
Studiengang: Master Informatik

Ludwig-Maximilians-Universität München

http://nbn-resolving.de/urn:nbn:de:bvb:19-epub-60621-8

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master Thesis

High Performance Dynamic Threading Analysis for Hybrid Applications

Felix Mößbauer

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

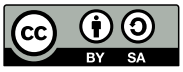
Master Thesis

High Performance Dynamic Threading Analysis for Hybrid Applications

Felix Mößbauer

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Tobias Fuchs (LMU) Dr. Christian Kern (Siemens AG)
Abgabe:	10. Januar 2019

Felix Mößbauer:
High Performance Dynamic Threading Analysis for Hybrid Applications
January 10, 2019



SUPERVISORS:
Tobias Fuchs (LMU)
Dr. Christian Kern (Siemens AG)

LOCATION:
Munich, Germany

TIME FRAME:
June 2018 - January 2019

Dedicated to my tortoise *Maxi*.
Slow and steady wins the race.

Declaration of originality

I hereby declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such.

Felix Mößbauer	City, Date

Abstract

Verifying the correctness of multithreaded programs is a challenging task due to errors that occur sporadically. Testing, the most important verification method for decades, has proven to be ineffective in this context. On the other hand, data race detectors are very successful in finding concurrency bugs that occur due to missing synchronization. However, those tools introduce a huge runtime overhead and therefore are not applicable to the analysis of real-time applications. Additionally, hybrid binaries consisting of Dotnet and native components are beyond the scope of many data race detectors.

In this thesis, we present a novel approach for a dynamic low-overhead data race detector. We contribute a set of fine-grained tuning techniques based on sampling and scoping. These are evaluated on real-world applications, demonstrating that the runtime overhead is reduced while still maintaining a good detection accuracy. Further, we present a proof of concept for hybrid applications and show that data races in managed Dotnet code are detectable by analyzing the application on the binary layer. The approaches presented in this thesis are implemented in the open-source tool DRace.

Keywords: Concurrency Bugs, Race Condition, Program Analysis, Binary Instrumentation, Sampling, Managed Applications

Acknowledgements

I would first like to thank my thesis advisor Tobias Fuchs at Ludwig-Maximilians-Universität München for his outstanding support and caring attitude throughout the course of this thesis.

I sincerely thank all the wonderful people in our lab at Siemens AG Corporate Technology for their inspiration and fruitful discussions. This work would never have been possible without the passionate participation and continuous support of my advisor Dr. Christian Kern, Dr. Matous Sedlacek and Dr. Tobias Schüle.

My deepest gratitude goes to my parents and to Melanie for providing me with unfailing support and continuous encouragement throughout the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

1 Introduction

The processing capabilities of a single core have increased only slightly over the past years, as a physical barrier has been reached. This drives chip manufactures to increase the number of cores per Central Processing Unit (CPU) to meet the demand for more powerful systems[Sut05]. At the same time, practice has shown that software applications have an ever growing need for computational resources while evolving. To follow this trend, developers are forced to write highly concurrent software, which is complex and error-prone. For legacy applications that were intentionally designed for a sequential system, concurrency cannot be easily introduced due to the tightly coupled components of the program.

A common starting point in this setting is to focus on the parallelization of computational intensive parts. However, often the internal dependencies are not well understood, e.g. interfaces between components have been bypassed in previous development cycles to improve the performance. This leads to concurrency related bugs in the parallelized version, which are notoriously hard to find as their occurrence depends on the runtime scheduling of the application. Neither unit nor integration tests are capable to reliable detect these bugs.

To narrow this gap, tools are required which assist the developer in finding and fixing these issues. Ideally this analysis could be done at compile time, but this is either to computationally complex, or not possible as the applications are first assembled at runtime. Furthermore, complex build environments and code that is written in many different programming languages inhibit this approach.

Other categories of software are programs written in managed programming languages which are executed under the management of a Common Language Runtime (CLR) virtual machine (e.g. Dotnet). These programs contain a Just-in-Time Compiler (JIT) compiler, which generates program code on the fly. While the analysis of the source or intermediate code of these programs is possible, this only holds for purely managed programs. Hybrid applications consisting of managed and native (processor-specific) code are beyond the reach of tools that analyze a binary while it resides on the disk.

Given that, a natural approach to inspect this kind of software is to analyze the application during execution. This makes it possible to analyze hybrid applications that consist of managed (JITed) and native code in a uniform manner. Transitions between both types of code are seamless and cross-module dependencies can be captured. Additionally, no adjustments like re-compiling or re-linking have to be made to the target application.

1.1 Problem Statement

A major drawback of state-of-the-art data race detectors is the large runtime overhead. This makes it impossible to analyze applications which are bound by external time constraints, e.g. real-time applications. Additionally, annotation of custom synchronization logic is not possible in many tools, which leads to false positive reports. Due to these reasons, a data race analysis is unfeasible in many use-cases.

For managed code applications, as well as for the Windows Operating System (OS), there is currently only a single dynamic data race detector available. However, the tool is proprietary and neither tuning is possible nor custom detection logic can be added. During tests on large applications the tool regularly crashes due to unknown reasons. Additionally, the runtime overhead is too high and annotations are not possible. Support for hybrid applications (native and Dotnet code) is available but based on approximations of synchronization techniques, which add additional false positives.

The goal of this thesis is to create a framework for runtime data race detection that meets the following requirements:

- **Customizable Instrumentation**
  The user should be able to specify which sections of the application under test will be instrumented. Custom synchronization logic can be implemented easily, both internally and externally by annotating the source code.

- **Adjustable Detection Accuracy**
  The tool should be adjustable to meet external time constraints while maintaining a reasonable race detection accuracy. Thereto, sampling and scoping techniques should be provided, both on a static configuration and on auto-tuning.

- **Hybrid Application Support**
  The tool should support hybrid applications consisting of native and Dotnet code (proof of concept). Data races between native and managed parts should be detected.

- **Exchangeable Race-Detection Algorithms**
  The framework should provide an interface to connect various race detector implementations to the instrumentation tool. The interface should be generic and closely map to the OS threading and memory management libraries.

Race detection algorithms itself are not part of this work. Here, we rely on an implementation of the commonly used LLVM *ThreadSanitizer*, which is attached through the detector interface. To add our instrumentation to the target application at runtime, we use the DynamoRIO runtime instrumentation framework.

The scope of this thesis lies on the instrumentation part and the development of novel techniques for both dynamic and static scoping.

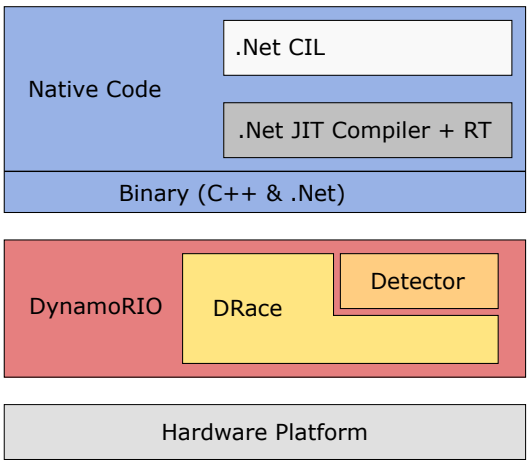


Figure 1.1: Layered architecture showing a hybrid application running under DRace.

1.2 DRace

In this thesis we present *DRace*, a modular runtime data race detection framework for Windows, implemented as a DynamoRIO tool (client). The framework targets native and managed (Dotnet) applications and provides a set of tuning techniques to minimize the runtime overhead. This includes sampling, scoping and auto-tuning techniques.

Custom synchronization logic can be easily encoded either directly into the detector, or by using the provided code-annotation mechanisms. Race detection algorithms are attached by implementing a generic interface. The live-reports of the detector are passed back to DRace, where they are symbolized and refined uniformly. The race summaries are presented in human readable format and can be exported to a Valkyrie compatible XML format.

DRace is available to the public in source code and binary form. It has been successfully tested on common open-source applications and a large industrial C++ application at the Siemens AG. A proof of concept for hybrid applications, containing Dotnet managed code based on the Dotnet Core runtime, is implemented as well.

1.3 Contributions

Correctness analysis of concurrent programs is a mature field of research. There are various approaches targeting different operating systems, programming languages and goals. Many modern approaches are developed for very specific use cases, as compared in Chapter 3. However, most of these systems are not applicable on Windows and only support native source code.

We contribute to this in multiple ways: by providing a framework for data race detection we enable developers to perform this analysis on programs which are beyond the scope of off-the-shelf tools due to the limitations listed above. Customizable detection logic and

annotations expand the field of application (Chapter 4). To achieve the previous goals, we introduce a technique to efficiently collect callstacks on Windows without using system calls, which is discussed in Section 4.4.

By analyzing the application on the binary layer, we show that native and managed code can be analyzed uniformly. Moreover, we show that data races in managed code are detectable by just analyzing the generated machine-level code. Based on this prerequisite, data races between managed and native code become visible. We further demonstrate a proof of concept on how to instrument Dotnet synchronization functions which behave purely in the managed domain. An example for this is the `System.Monitor` object, which uses a Spinlock and falls back to a native lock only if spinning takes too long (Chapter 5).

We show that various sampling techniques can be applied to race detection to limit the overhead, while recurring data races are likely to be identified even for long sampling periods. This brings us a race detector where the tradeoff between overhead and accuracy can be controlled. Finally, this enables us to meet external time constraints. We discuss this in more detail in Chapter 6.

In Chapter 7 we evaluate the framework on real world applications and demonstrate that our tool can be tuned to significantly reduce the runtime overhead during the analysis. Finally, we revisit the objective of this thesis and conclude our findings in Chapter 8.

1.4 Definitions

Definition 1.4.1 (Race Condition)**.** A race condition denotes a situation where the output of an operation depends on the chronological order of other uncontrollable events[Huf54].

Definition 1.4.2 (Data Race)**.** A data race is a situation where two threads concurrently access a shared memory location and at least one of the accesses is a write.

Definition 1.4.3 (Segment)**.** A sequence of events of one thread that contains only memory access events (e.g. no synchronization events)[SI09].

Definition 1.4.4 (Sampling)**.** From a sequence of elements, approximately each T's element is drawn. T denotes the sampling period.

Definition 1.4.5 (Module)**.** A library or executable that contains machine-code instructions (e.g. a Dynamic Link Library (DLL)).

2 Background

This chapter defines the scope of this thesis and locates it in the surrounding field of application. We categorize our work under four major domains, classify our contribution in the context of existing tools and provide information on further research. A conceptual visualization of the domains is provided in Figure 2.1.

2.1 Software Verification and Validation

Ensuring the correctness of software has always been a major area of research in computer science. In this context various strategies evolved, mainly differing in a tradeoff between the effort to perform the analysis and the universal validity of its results.

In its simplest form, the input/output behavior of a program is tested. This means, a function or the whole program is called with a predefined input and the result is then compared to the known (e.g. manually computed) output. When this analysis is performed on single functions or small parts of a code base, it is called *Unit Testing*. To support the user in writing these tests, various frameworks like GoogleTest and Catch2[Nas17] have been developed. As it is in general not possible to test all input-output pairs, the results can only falsify the correctness of the component. This is the case if an output differs from the expected one.

Issues related to the concurrent execution of a program are often not found using unit tests, as they depend on the schedule of the execution. Finding these non-deterministic bugs is the scope of *Dynamic Threading Analysis*. Here, two common problems are deadlocks and data races. While deadlocks prevent the further execution of a program, data races often lead to incorrect results and application crashes. Common tools to find these issues during the execution of the program are the ThreadSanitizer (TSan)[SI09], Intel® Inspector XE and Helgrind. A general issue of data race detection tools is the considerable slowdown of the target application during the analysis.

Dynamic Threading Analysis tools are only capable of locating these bugs to a certain extent, which is substantiated in Section 3.3. This also applies to our Dynamic Threading Analysis tool, presented in this thesis. If stronger guarantees on the correctness are required, Static Model Checking approaches can be used. These analyze all possible execution paths of the source code and validate them against a model. Due to the nature of this problem, a complete analysis of all paths implies extreme computational cost. Some tools like the

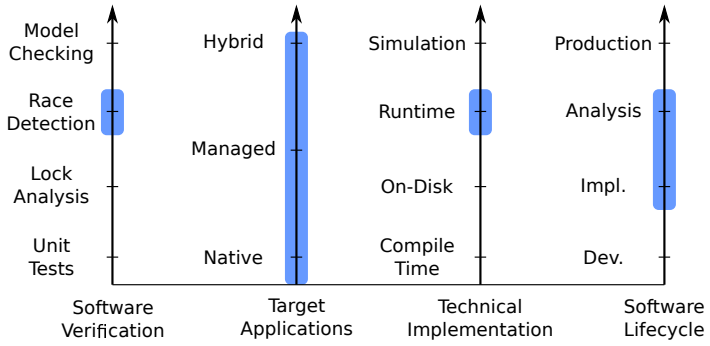


Figure 2.1: Scope (blue) of this thesis located in the surrounding field of applications.

Java Pathfinder[HP00] (for Java applications) perform this exhaustive search and prove the correctness of the code according to the model. This requires all states to be analyzed at the cost of a long analysis time. Other tools like the HSR-Parallel-Checker[Blä18] (for C# applications) use heuristics to provide a good tradeoff between the analysis time and the quality of the results.

Finally these tools only verify that the source code behaves according to its specification, but do not verify the compiler and the system the software is running on. To verify the machine language program of a verified source program, the whole tool chain in between has to be verified. This is accomplished by the Verified Software Toolchain[App11] project for C-language programs.

2.2 Target Applications

Runtime instrumentation tools are always developed for a specific target and system architecture. In this context, we differentiate native and managed applications whereas the transition between both classes is smooth. With native applications we refer to machine code that is static, i.e. it does not change during the execution. In contrast managed codes are stored in an intermediate format and generated at runtime by a JIT.

For purely managed applications, the runtime instrumentation can be performed directly in the managed language. Well-known representatives of this approach are the RoadRunner dynamic analysis framework[FF10] for java applications and `Mono.Cecil`[Xam08] for Dotnet based applications. To the extent of our knowledge, the only tool that is able to analyze hybrid applications consisting of native and Dotnet parts is the Intel® Inspector XE.

Our tool DRace introduces a novel concept to analyze hybrid applications uniformly by considering the just-in-time compiled machine-language code along with the native parts. At this, with managed code we refer to code that is based on the Dotnet framework. Analysis of other managed programming languages like Python and Java are not part of this thesis.

2.3 Technical Implementation

The implementation of a system for Dynamic Threading Analysis specifies how to obtain the data that is required to perform the analysis. Most analysis tools consist of a frontend and a backend (runtime): the frontend specifies the component that obtains the analysis related data. This code (instrumentation) then communicates with the backend to perform the analysis.

One option is to add instrumentation code directly to the target application during compilation. A tool that follows this approach is the TSan, which uses the LLVM compiler infrastructure to add the instrumentation.

For situations where either no suitable compiler is available or the target application is already compiled to a binary, runtime instrumentation tools like Pin[Luk+05], Valgrind[NS07] and DynamoRIO[Bru04] have been developed. Our tool DRace is also based on this approach. While these tools are not analysis tools itself, they modify the machine code during execution and provide an interface to add the instrumentation. If a modification of the target application is not possible at all, simulators can be used. For a complete system simulation, frameworks like MARSS[Pat+11] (x86 only) and SIMICS[Mag+02] (generic) are available.

A different approach is to use the CPU hardware counters and the performance measurement unit in combination with modified threading libraries. These counters can be accessed using tools like PAPI[Muc+99] and likwid[THW10]. The applicability of this technique has been shown by the data race detector RACEZ[She+11].

2.4 Employment in Software Lifecycle

Correctness analysis tools are involved in various steps during the development cycle of an application. Which tool is applied in a step is mainly affected by the added runtime overhead and the degree of automation. With the choice of the tool, aspects like the reliability of the analysis results are determined. Here it is necessary to make a tradeoff between the overhead, degree of automation and the generality of the results.

During the development of algorithms and protocols, theorem prover like Isabelle[Pau94] and Simplify[DNS05] are used to prove the correctness of the design. This step requires in-depth knowledge and is difficult or impossible to automate.

To verify the implementation of an algorithm against a model, software model checkers like SPIN[Hol97] and BLAST[Bey+07] are applicable. These tools are computationally expensive and require some manual assistance, hence they have to be applied during development or in dedicated analysis steps. After some initial tuning, Dynamic Threading Analysis tools like TSan[SI09], Helgrind and Intel® Inspector XE can be applied in automated tests. Likewise, our tool DRace is intended to be used in this setting.

Finally, tools have been developed specifically for being used during production. A data race detector in this setting is RACES[She+11] which uses a probabilistic approach for detection and is intended to be used on long running server processes. Additionally, extensive hardware support is required which limits the scope of this tool to a specific environment.

3 Related Work

With the availability of the first multi core systems in the early 1990, a new area of research evolved: finding correctness issues in parallel programs. A variety of algorithms and software regarding this topic has been developed.

This Chapter first describes state-of-the-art algorithms used to find concurrency issues. After that, we examine a variety of existing tools for data race detection.

3.1 Algorithms

Mainly two types of algorithms are used for data race detection: on the one hand, algorithms based on a lockset and on the other hand algorithms based on Lamport's Happened Before relation.

3.1.1 Lockset

Lockset algorithms analyze the locking discipline of all threads. A common lockset based algorithm is Eraser [Sav97], developed in 1997. The key idea is to monitor active locks (e.g. mutexes) that are held during an access to a memory location by the current thread. Locations accessed by at least two different threads where at least one access is a write are data race candidates. For these candidates all locks which the accessing thread currently holds are put into the lockset. After that, for every other access to this location, the intersection of both locksets is computed. If the intersection is empty, a potential concurrency error is found.

Algorithm 1 Basic version of the lockset algorithm used in Eraser [Sav97][p.396]

Let $locks\_held(t)$ be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t
  set $C(v) := C(v) \cap locks\_held(t)$
  if $C(v) = \emptyset$, then issue a warning

The authors show that this algorithm is easy to implement and provides good detection results. However, it is only suited for pure lock-based programs, as this is the only synchronization mechanism which is tracked.

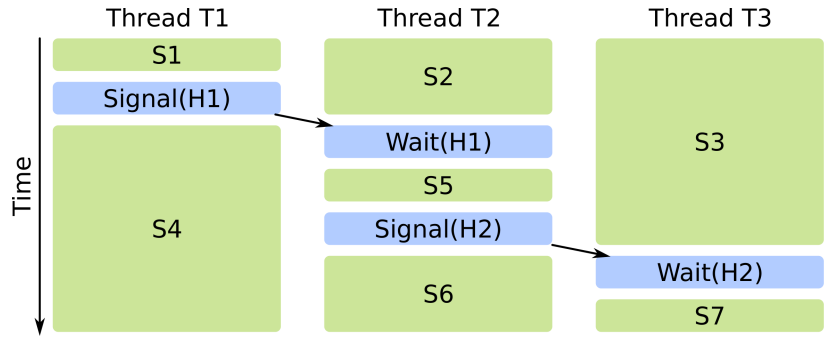


Figure 3.1: **Example of the Happened Before relation.** $S_1 \prec S_4$ (same thread); $S_1 \prec S_5$ (happens-before arc $Signal_{T_1}(H_1) - Wait_{T_2}(H_1)$); $S_1 \prec S_7$ (happens-before is transitive); $S_4 \not\prec S_2$ (no relation).[SI09, p.63]

Future implementations of the algorithm focus on both reducing the number of memory accesses that are tracked, as well as accelerating the computation of the locksets. One optimization is to drop local only accesses. This can be done by grouping accesses into cache lines and only tracking the lockset for each line. When the first shared access to this line is detected, the entries are separated and tracked individually. While this approach slightly reduces the accuracy, it increases the performance by up to factor two [SI09][4.4.2].

3.1.2 Happened Before

Happened Before (HB) based algorithms analyze the causal order of events in an asynchronous system. Lamport formalized the concept of one event happening before another in "Time, Clocks, and the Ordering of Events in a Distributed System" (1978)[Lam78]. This causal order is defined by the Happened Before relation:

Definition 3.1.1. Two events a, b are in the happened before relation iff the following conditions are met. Here, $T(x)$ denotes the (logical) local time of event x.

1. On the same process, $a \prec b$ if $T(a) < T(b)$

2. If a sends a message to b then $a \prec b$

Concurrency errors are defined as pairs of unordered events, with respect to this relation. The Happened Before relation can be constructed while the application executes (on-the-fly). As the relation is transitive, irreflexive and antisymmetric, it defines a strict partial order. Algorithms which check if two elements are in this relation utilize the transitivity property of the relation to reduce the computational complexity.

As message we consider $Signal \rightarrow Wait$ sequences between two threads. In the case of mutexes, this directly maps to a `Release` call followed by an `Acquire` call.

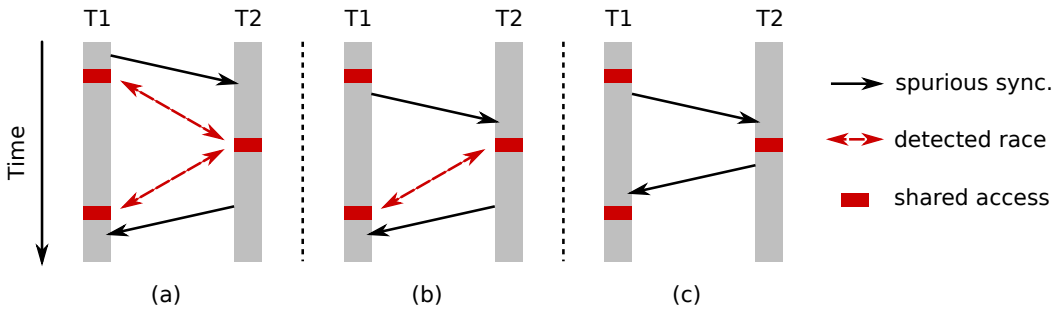


Figure 3.2: In pure Happened Before based detectors, the timing of spurious synchronization events determines if a data race is found.

The relation can be represented as a Directed Acyclic Graph (DAG). In the event graph, a message is pictured by an edge between a `Signal` event and a `Wait` event on two threads. An example of this graph is provided in Figure 3.1.

A key aspect of Lamport's work is the differentiation between global and local time stamps. Herewith, only a minimal overhead is required to determine a partial order of events. This is later extended to the more advanced concept of vector clocks which enables a tracking of causal events. We refer the interested reader to [Tv07][p.248] for more details on this topic.

Naming While the original work [Lam78] names the relation described in Def. 3.1.1 "Happened Before", several later works like [SI09], [Atz+16] and [PS03] use the term "happens-before". In this thesis, we prefer the former naming. Nevertheless, when we refer to a specific related work, we use the corresponding denotation.

Limitations In practice, it is too expensive to maintain a vector clock per shared memory location. Hence, implementations have to group accesses or drop far back events to limit the memory consumption. The Happened Before algorithm is prone to unintended synchronization events due to spurious inter-thread messages. By that, actual concurrency bugs are missed as the related HB-edges feign calls to synchronization procedures [OC03][p.171]. Figure 3.2 visualizes this issue for three different schedules.

3.1.3 Hybrid Algorithms

Pure lockset based detectors are not that common anymore, as synchronization methods beside locks are not supported. This leads to false positives on programs that use other mechanisms, e.g. lock-free data structures.

O'Callahan et al. showed in [OC03] that the lockset approach is not suited to modern programs which use a variety of synchronization mechanisms. They present a new algorithm which uses both locksets and happended-before based detection. The idea of combining both approaches has already been discussed earlier in [DS91], but has not been implemented prior

to [OC03]. This reduces the shortcomings of both algorithms which leads to a faster and more accurate data race detection system.

This reduces the large overhead of pure happend-before algorithms while also reducing the number of false positive detections obtained by a lockset algorithm. As only a limited form of Happened Before detection is used, fewer concurrency bugs are hidden due to spurious synchronization.

3.1.4 Approximative Algorithms

Most data race detectors are based on locksets and Lamports Happened Before relation. The main improvements in recently developed algorithms are strategies to compute good and fast approximations of these methods.

In 2015 a new algorithm called "FastTrack" was presented by Cormac Flanagan et. al which provides better scaling in the number of threads which are analyzed [FF09]. Their main achievement was a formalized approximation of the Happened Before relation which can be computed in $O(1)$ instead of $O(n)$ in the number of threads. We refer the interested reader to [FF09, p.3] for details on this optimization. The FastTrack algorithm and its successor FastTrack2 [FF17] are implemented as part of the RoadRunner data race detector for Java applications. The main improvements in FastTrack2 are refined analysis rules which result in a more straight forward implementation of the algorithm.

Vector clock based algorithms are known for its precise detection at the cost of a high runtime overhead. For object oriented programming languages this is not an issue as the detection takes place at object granularity. As this is not sufficient for C/C++ programs, Song et al. present a novel approach to track accesses with a dynamic granularity. They implemented a data race detector on top of the FastTrack algorithm which has 43% less overhead in average compared to the unmodified version of FastTrack with a similar race detection quality[SL14].

In [MMN09] a sampling based approach was presented to further limit the overhead. The authors show that approximately 70% of all data races in common Windows applications can be found when sampling just 2% of all memory referencing instructions. While the presented algorithms are specialized to find data races during the development process, Sheng et. al implemented a tool which can be applied during production [She+11]. Their algorithm "Racez" uses a hardware based sampling technique to limit the runtime overhead to ≈10%. The main field of application are long running server processes, where message processing loops are executed millions of times. In addition the authors provide a stochastic estimation on the probability of finding a (present) data race. This approximation is parameterized on the number of executions and the sampling rate.

3.2 Related Tools

While the previous section focused on the algorithmic foundations of data race detection, this section covers tools implementing the tracking of synchronization and memory accesses.

A prominent system is the TSan [SI09] which was developed at Google™ and is now an integral part of Clang and the GNU Compiler Collection (GCC). It uses a hybrid detection algorithm combining locksets and Happened Before, similar to [OC03]. The instrumentation is added during compile time and is only available for Linux applications. While the primary focus of the TSan are C++ applications, it has been adapted for the programming language Go as well. We use the backend of the TSan implementation for Go, as it provides an interface to the native backend through a library. Verification tools from the scientific domain adapt this library to validate arbitrary concurrency semantics which can be described as Happened Before relation. This includes a correctness checker for the OpenMP model [Atz+16] as well as one for finding bugs in Message Passing Interface (MPI) applications [Hil+13].

The main limitation of the TSan is that it cannot be applied to already compiled binaries and is limited to 64bit applications. At that time, this makes it impossible to use it in scenarios requiring a complex build process and vendor-specific compilers. As we just rely on the race-detection backend of TSan and not the instrumentation part, these limitations do not apply to our tool DRace.

To our knowledge, there exist three dynamic instrumentation tools suited for data race detection at this time: Intel® PIN is a commercial framework to dynamically add instrumentation to a binary at runtime, also supporting Windows. The Intel® Inspector XE uses PIN and is shipped with a data race detector for C++11 and C# (Dotnet CLR). However, comprehensive tests have shown that the C# synchronization methods are not correctly processed. We assume that the tool internally uses a hybrid algorithm but regarding that there is only limited information publicly available.

A common binary instrumentation framework for Linux is Valgrind[NS07]. While the framework is mostly known for the memory-leak detection, it also contains a race-detector called Helgrind[MW07]. Internally it uses a hybrid algorithm and provides very extensive information on each race that is detected[JT08].

DynamoRIO is a tool similar to Valgrind, providing methods for binary instrumentation at runtime. The tool was developed by Derek Bruening as part of his PhD Thesis[Bru04]. It supports Windows as well as Linux on x86, x64 and arm architectures. While DynamoRIO itself does not provide a data race detector, we use the framework to add the instrumentation for our race detector DRace. In contrast to Intel® PIN, the implementation is completely open source which significantly simplifies debugging.

For the sake of completeness, there exist static correctness checking tools. These tools analyze the source code of a program. A recent tool for the C# programming language is the HSR Parallel Checker [Blä18]. However, it is unlikely that static analyzers will achieve good

results on large applications, possibly consisting of parts written in multiple programming languages. Dotnet applications often consist of managed-code parts for the application logic and native parts for computational intensive modules. The HSR Parallel Checker abstracts the native parts, resulting in an additional black-box state that is not analyzed.

3.3 Correctness

In theoretical computer science the terms sound and complete are used to describe correctness aspects of an algorithm.

Soundness describes that if the algorithm does return anything (in our case a data race), this result is correct. Thus, it does not detect a data race if the application is race-free according to the specification of the algorithm.

The completeness aspect defines that the set of results always contains all correct results, but possibly also incorrect ones. In our case this means all known-to-be existing data races are reported but possibly also additional (false positive) ones.

Both lockset and Happend Before algorithms are sound and complete. Hence all existing data races are detected, but no additional ones. However, this only holds for the algorithm and not the race detection tool, due to the following reasons:

- **Execution Paths**: a program-run might not traverse all execution paths. For instance, there are likely program sections which are never executed (e.g. Exceptions, Error-Handling,...)
- **Heap Accesses**: even if all code-lines are executed (measurable using code-coverage), not all possible access patterns on the heap might be traversed.
- **Custom Synchronization**: custom synchronization logic is unknown to the detection tool and hence false positives are detected.

In contrast to formal model checking techniques - which actually prove correctness - we can only falsify a given program. This means, if no races showed up, either our detector did not find any, or there are no races. These limitations of a dynamic race detection tool are discussed by Kahlon et al. in [KW10] in more detail.

In practice additional issues arise: only synchronization primitives which are implemented in the detector are correctly processed. This automatically leads to false positives if hand-crafted synchronization methods are used. Furthermore, both time and memory resources are limited, so the internal state of the detector used for race detection is limited. If the internal state exceeds some threshold, parts of the state have to be dropped or merged.

Summarizing, we specify the detection algorithm in a way that it detects data races with small false positive and small false negative errors. Various other research has shown that this is sufficient and leads to good results.

3.3.1 Benign Data Race

Data races which are not concurrency related bugs are called “benign” races. These type of races often occur in updates to statistic counters and implementations of synchronization procedures itself [KZC12, p.186]. Often these races are intentional to avoid the overhead of synchronization or atomic accesses when approximations of a value are sufficient (e.g. statistics counters).

While these races are not harmful and mostly intentional, they are reported by a data race detector. Kasikci et al. present a heuristic to distinguish between harmful and harmless (benign) data races [KZC12]. We decided to not implement this technique in DRace due to both complexity and possible miss classification. In our opinion benign races should be manually inspected by the user of the tool and either annotated or suppressed.

4 Architecture and Implementation

In this chapter we cover the architecture and the implementation of a tool that is capable to reliably detect all memory accesses and synchronization events to finally perform the race analysis. In this context, we aim to implement this tool with the lowest possible runtime overhead. Additionally, we have to keep the interference of the instrumentation and the race detector with the target application as small as possible (viewed from the targets perspective). Inconsiderate changes are likely to result in application crashes or deadlocks. This interference problem will be discussed in more detail in Section 4.3.

All modifications of the application under test are performed at runtime using the DynamoRIO framework, whereat DRace is registered as a "client". In this connection, DRace registers a set of functions (callbacks) which are called by DynamoRIO on certain events. This enables the client e.g. to act on module loads and to inspect or modify the instructions that are going to be executed. The internal architecture of DRace is visualized in Figure 4.3. The respective modules of DRace are described in the next sections. For further details on the interaction of DynamoRIO and its client, we refer the interested reader to [Bru18].

4.1 Event Types

All information that is fed into the detector is called an event. We hereby differentiate between mandatory and optional events. They are classified by the impact on the correctness of the race detection, when the event is missed.

Definition 4.1.1 (mandatory event)**.** If a mandatory event is missed, non-existing data races might be detected (false-positive).
Note: All synchronization procedures are covered by this class.

Definition 4.1.2 (optional event)**.** If an optional event is missed, data races might be missed (false-negatives).
Note: All memory accesses are covered by this class.

4.2 Generic Interface

Instead of directly passing the events to the detector, we use an intermediate interface. This enables a clear separation between the event producer and consumer. While the instru-

mentation part remains static (i.e. it does not change for other detectors), we can plug in different race detection algorithms.

The interface is specified in a header file and is attached using dynamic linking at starting time. Using static linking is not possible due to the limitations of the private loader in DynamoRIO.

When a race is found, the detector backend has to notify DRace about it. This is done utilizing a callback method which is registered in the `detector::init()` method. To get information on the race, a pointer is passed as argument to the callback. By this, the race is further processed in DRace.

To get a clean separation between DRace and the detector backend, no non-race detection related information is passed to the detector. This means, the symbol lookup of instruction pointers is done in DRace and not in the detector.

We do not make any assumptions on when the callback is called. They might also be called concurrently.

Data Types

We specified common data types for a unified communication with the detector backend. This is necessary for the Application Binary Interface (ABI) of the detector.

A data race is characterized by a tuple of two accesses. Each access contains information on the memory location, write-mode, callstack, thread and memory block (if any) and is stored in the `AccessEntry` struct. To avoid dynamic allocation, we use a fixed size buffer in the struct for storing the callstack. The `AccessEntry` is constructed internally in the detector and passed back to DRace in the callback when a data race is detected. To push events into the detector, the corresponding function of the interface are used.

While all addresses (memory location, callstack entries) are `void` pointers, we store them as 64-bit integers to make clear that these values are just arbitrary addresses.

Listing 4.1: data race information

```
struct AccessEntry {
  unsigned thread_id;
  bool     write;
  uint64_t accessed_memory;
  size_t   access_size;
  int      access_type;
  uint64_t heap_block_begin;
  size_t   heap_block_size;
  bool     onheap;
  size_t   stack_size;
  uint64_t stack_trace[max_stack_size];
};

/* A Data-Race is a tuple of two Accesses*/
using Race = std::pair<AccessEntry, AccessEntry>;
```

Functions

Before passing in any events, the detector has to be initialized using `init(...)`. After `finalize()` is called, no further function calls are allowed (except `init(...)`). This is required as the detector must not be stateless. The events are then passed to the detector using functions.

Listing 4.2: startup and teardown of the detector

```
/* Takes command line arguments and a callback to process a data race.
* Type of callback is (const detector::Race*) -> void
*/
bool init(int argc, const char **argv, Callback rc_clb);

/* Finalizes the detector.
* After a finalize, a later init must be possible.
*/
void finalize();
```

Each data race record contains two callstacks. Instead of passing a complete callstack on each operation, we track the call and return statements. This enables a much more efficient implementation of the callstack tracking in the detector. Additionally, this also reduces the amount of data that is moved between DRace and the detector backend.

Listing 4.3: callstack tracing

```
/** Enter a function (push program counter to stack) */
 void func_enter(tls_t tls, void* pc);

/** Leave a function (pop stack entry) */
void func_exit(tls_t tls);
```

Most race detectors keep internal thread IDs / states for the mapping of physical threads to logic accesses. This means, the thread ids obtained by the OS have to be mapped to the internal ones for each call. While this could be done using a small hashmap, we specify the interface directly in a way which avoids this: Each thread provides a tiny memory buffer of 64 bit in its Thread Local Storage (TLS) space which can be modified by the detector. This buffer is initialized in the `fork(...)` event and passed in each further call to the detector.

Listing 4.4: thread events

```
/** Log a thread-creation event */
void fork(
  tid_t parent, /// id of parent thread
  tid_t child,  /// id of child thread
  tls_t * tls   /// out parameter for tls data
);

/* Log a thread join event */
void join(
  tid_t parent,
```

```
  tid_t child,
  tls_t tls
);

/* Log a thread detach event */
void detach(
  tls_t tls,
  tid_t thread_id
);

/* Log a thread exit event (detached thread) */
void finish(
  tls_t tls,
  tid_t thread_id
);
```

Synchronization events can either be lock-based or pure Happened Before (HB) based. For pure HB detectors, lock-based events directly map to the HB logic. However, with this separation additional errors like double locking can be detected.

We support both techniques and recommend the user to use the lock-based logic when instrumenting locking mechanisms. Additionally, we support exclusive locks, reader-writer locks and recursive locks. If a lock is tested (try-locked), the corresponding `detector::aqcuire` must only be called if the lock has successfully been acquired.

Both for locks and happened before, the identifier can be either a memory address or a handle. However, the identifier must be unique for this lock. This might not be the case for handles.

Listing 4.5: synchronization sematics

```
/* Acquire a mutex */
void acquire(
  tls_t tls,       /// ptr to thread-local storage of calling thread
  void* mutex,     /// ptr to mutex location
  int   recursive, /// number of recursive locks (1 for non-recursive mutex)
  bool  write      /// true, for RW-mutexes in read-mode false
);

/* Release a mutex */
void release(
  tls_t tls,       /// ptr to thread-local storage of calling thread
  void* mutex,     /// ptr to mutex location
  bool  write      /// true, for RW-mutexes in read-mode false
);

/* Draw a happens-before edge between thread and identifier (optional) */
void happens_before(tid_t thread_id, void* identifier);

/* Draw a happens-after edge between thread and identifier (optional) */
void happens_after(tid_t thread_id, void* identifier);
```

For each memory access, a callstack is recorded as well. If a race occurs on this memory address, the callstack is passed in the `AccessEntry` of the race information. To limit the overhead of the detector, callstacks should be as small as possible. The size of the access is specified in $\log_2(bytes)$. So an access of 2 bytes results in $size = 1$.

Listing 4.6: memory access events

```
/* Log a read access */
void read(
  tls_t    tls,       /// ptr to thread-local storage of calling thread
  void*    pc,        /// current program counter
  void*    addr,      /// memory location
  size_t   size       /// access size log2 (bytes)
);

/* Log a write access */
void write(
  tls_t    tls,       /// ptr to thread-local storage of calling thread
  void*    pc,        /// current program counter
  void*    addr,      /// memory location
  size_t   size       /// access size log2 (bytes)
);
```

Memory blocks are tracked to reset the memory range on a subsequent deallocation. The interface is modeled according to the semantics of `HeapAlloc()` and `HeapFree()`, so that the deallocation call does not know the size of the block.

Listing 4.7: memory allocation and deallocation

```
/* Log a memory allocation */
void allocate(
  tls_t  tls,       /// ptr to thread-local storage of calling thread
  void*  addr,      /// begin of allocated memory block
  size_t size       /// size of memory block
);

/* Log a memory deallocation*/
void deallocate(
  tls_t tls,       /// ptr to thread-local storage of calling thread
  void* addr       /// begin of memory block
);
```

4.2.1 Custom Annotations

Dynamic race detectors have to know the synchronization mechanisms used by the program under test. Otherwise the detector will not work. For programs that only use POSIX synchronization, the underlying logic can be hard-coded into the detector. However, if the program uses additional synchronization techniques, false positives will occur.

For example, many modern programs use a combination of POSIX mutexes and fast user-level Spinlocks to avoid the overhead of system calls. For this purpose we provide a set of

annotation macros. These have to be inserted by the developer of the target application into the source code. The annotation macros are expanded into empty functions (marked as "do not optimize away") which are then intercepted by the instrumentation tool.

The macros are named according to the ones used in TSan [SI09]. The most important ones are:

- `ANNOTATE_HAPPENS_BEFORE(ptr)`
- `ANNOTATE_HAPPENS_AFTER(ptr)`

These annotations can also be used to annotate lock-free synchronization.
Other annotations include:

- `ANNOTATE_ENTER_EXCLUDE`
- `ANNOTATE_LEAVE_EXCLUDE`

These are used to exclude known-to-be race free regions and code that has benign races.

Example One-to-one message based synchronization (two participants) is annotated as demonstrated in Listing 4.8. Here, the receive invocation blocks until the corresponding send has been fully received. This creates a Happened Before relation between all memory-accesses prior to `send` and after `recv`. Hence, we place the "happens-before" annotation prior to the signal (`send`) to ensure that the detector is informed before the corresponding `recv` finishes. The "happens-after" annotation is placed after the `recv` statement, as this call has to return prior to the event. Otherwise, memory accesses of the `recv` function itself would be mapped to the wrong segment.

The Happened Before relation always belongs to an unique identifier. If a mutex is used for the synchronization, its memory location is suited. In the situation described in Lst. 4.8 no natural identifier is available. Hence, we use an arbitrary id (`0x42`).

Listing 4.8: Annotation of a message based synchronization using the Happened Before logic.

```
#define DRACE_ANNOTATION // Enable Annotations
#include <drace_annotation.h>

// [...]
if(myid == 0){
  ANNOTATE_HAPPENS_BEFORE(0x42) // event id
  send(msg, 1); // send to rank 1
}

if(myid == 1){
  recv(msg, 0); // recieve from rank 0
  ANNOTATE_HAPPENS_AFTER(0x42) // event id
}
```

4.2.2 Interface Implementation

We implement this interface around the backend of the TSan which has been separated for the programming language "Go". As there is currently only a version of the detector which uses the Linux/Unix Application Programming Interface (API), we compile it using MinGW and link into the DLL. To get the symbol mangling right, a slim C wrapper is used around the C++ interface of the TSan.

As TSan's interface does not fully match our generic interface, we have to track some state. This includes the size of allocations, as the corresponding deallocation function does not provide this information. To clearly separate the instrumentation from the detector logic, this is done in the implementation of the interface.

4.3 Instrumentation

With instrumentation we refer to our modifications of the target application to intercept race detection related events. As all modifications of the application under test are performed in memory, the executable residing on the disk is not changed.

To fully understand the following section, basic knowledge about computer architecture and the Assembly programming language is required. For those who are not familiar with this, we refer to [TA13, p.701].

4.3.1 Technical Foundations

A computer program (binary) consists of data and a list of instructions. These are mapped to an address in the virtual logical address space of the executing system. The Instruction Pointer (IP) (also called program counter) points to the next instruction that is going to be executed. After the execution of an instruction, the IP is incremented by one and hence points to the next instruction. If the instruction is a Control Transfer Instruction (CTI) (e.g. `JMP`, `CALL`, `RET`, ...), the IP is set to an address specified by the operand of this instruction[TA13, p.54].

Addressing Modes After a CTI has been executed, the program continues at an arbitrary location. This address is either specified by an offset from the CTI (relative addressing) or by using an absolute address (absolute addressing). One special case of absolute addressing is Load-Time Locatable (LTL) code that is modified by the linker / loader to be run from a particular memory location.

Code Cache To be able to add instrumentation to the target, instead of direct execution, the application is executed from a code cache. This is necessary as all instructions behind a modification in the instruction list that changes its length have to be shifted. This includes re

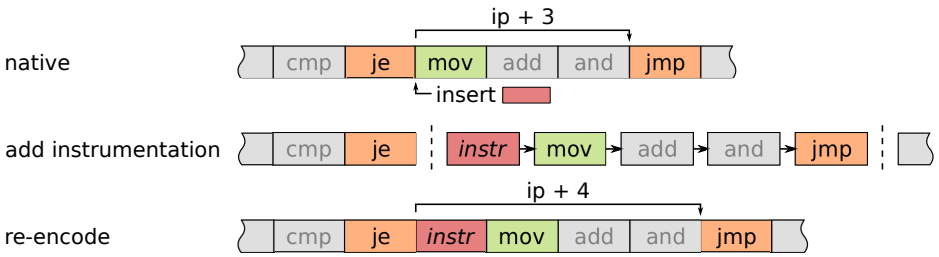


Figure 4.1: Process of inserting additional instructions into the instruction list. Orange blocks represent CTIs, the `mov` instruction is going to be instrumented.

calculations of the addresses of jumps as well. Figure 4.1 visualizes this scenario for a single instruction that is inserted in front of the `mov` instruction. After adding the instruction, the re-encoded block is stored in the code cache.

At this juncture, the code cache is a memory block allocated on the heap which is marked as executable and contains a translated copy of the application's instructions. The translation step then modifies the addresses of CTIs to point to the corresponding ones in the code cache. Translation also applies to code that uses relative addressing as the distance between both instructions changes if code is inserted or removed in between. The translation is done at the granularity of fragments, where a fragment is defined as a sequence of instructions that terminates with a control transfer operation.

4.3.2 Instruction Injection

When a code fragment of the target is going to be executed, it is first translated into the code cache and then executed from there. At this time the client (DRace) is notified via the "basic block" event and analyzes and modifies the fragment that is put into the cache.

DynamoRIO provides mechanisms to disassemble the fragment and to add additional assembly code (or change existing). This is called "inline instrumentation", as the additional assembly is directly inserted into the fragments. While this is the most efficient way to add instrumentation, this becomes uncontrollably complex for sophisticated logic. To solve this issue, DynamoRIO provides a mechanism to add calls into client functions (DRace is registered as DynamoRIO client). This is called a *clean-call*, as DynamoRIO takes care of preserving the current application state and restoring it after the call.

Stateful Instrumentation For certain requirements, the instrumentation has to behave according to some internal state of DRace. This is accomplished through a Thread Local Storage (TLS) buffer, which can be accessed from inline instrumentation, the code cache and the client. The TLS is registered and initialized in the "thread-start" event and remains valid until the "thread-end" event.

We use this TLS to preserve parts of the application state during the inline instrumentation, as well as for controlling DRace.

Transparency As the instrumentation is executed along with the target application, they also share a common state. For some values like CPU registers and floating point calculations, DynamoRIO provides mechanisms to preserve and restore them. However, there are also limitations on what the client must not use: this includes exceptions, system mutexes, and some syscalls. Here, it is to be noted that this also holds for all libraries which the client loads. Additionally, caution has to be paid when using locks. As both client and target application locks interfere, deadlocks might happen.

A non-trivial transparency issue we discovered during our tests with various applications was when we tried to share the stack with the target application. Here the idea was to reduce the number of registers required for the memory reference instrumentation by temporary pushing data to the application stack and removing it before the application continues to execute. However, some applications place data above the top of stack pointer value. Hence, we overwrote this data with our `push` instruction and later on the target application crashed.

4.3.3 Memory Reference Instrumentation

Instead of directly passing the memory accesses to the detector, we buffer them in a thread-local buffer and pass them in the next *clean-call*. To ensure the correctness of the detection, each synchronization event has to trigger a *clean-call*. In Theorem 4.3.1 we show that this procedure is correct regarding the Happened Before logic. For lockset-based detectors this holds as well, as all accesses of the local buffer belong to the same segment and hence share the same lockset.

Theorem 4.3.1 (local buffer)**.** Let s_A, s_B, S be a sequence of events and A, B distinct Threads ($A \neq B$).

- $s_A = [a_1, \text{HB}, a_2]$: a_i memory access, HB Happened Before event on A,
- $s_B = [b_1, \text{HA}, b_2]$: b_i memory access, HA Happened After event on B

$S = [s_A \cup s_B]$: local order is preserved, HB $\prec$ HA

Claim: The result of the Happened Before relation is equal for all S, i.e. if one pair is (not) in relation in one sequence it is also (not) in relation in the other sequences. *Note:* In practice we only consider pairs with a shared memory location but this is no constraint of the theorem.

Proof. The Happend Before arc HB $\rightarrow$ HA leads to the following global sequences:

1. $[a_1 \cup b_1, \text{HB}, \text{HA}, a_2 \cup b_2]$
2. $[a_1 \cup b_1, \text{HB}, a_2, \text{HA}, b_2]$

3. $[a_1, \text{HB}, b_1, \text{HA}, a_2 \cup b_2]$
4. $[a_1, \text{HB}, a_2 \cup b_1, \text{HA}, b_2]$

Let $i, j \in \{1, 2\}$. According to the definition of a_i and b_i: $a_1 \prec a_2$, $b_1 \prec b_2$ and $a_2 \not\prec a_1$, $b_2 \not\prec b_1$. Additionally, $b_i \not\prec a_j$ always holds as there is no direct (non-transitive) relation between thread A and B except for HB $\prec$ HA (*Note:* consider scenario $S_4 \not\prec S_2$ in Fig. 3.1).
Remaining cases:

- (a_1, b_1): then in all of the four cases $a_1 \not\prec b_1$ since $a_1 \prec \text{HB} \prec \text{HA}$ but $b_1 \prec \text{HA}$.
- (a_1, b_2): then $a_1 \prec \text{HB} \prec \text{HA} \prec b_2$ (transitive).
- (a_2, b_1): then $a_2 \not\prec b_1$ since $a_2 \not\prec \text{HB}$. Analogously for $a_2 = b_2$.

□

To trace the memory that is accessed by the target application, we instrument each memory referencing instruction. Here, we determine the following properties:

- source / target address
- size of access
- instruction pointer
- read / write mode

The tracing is done using highly-tuned inline instrumentation, as shown in Listing 4.9. The access-entry is stored in a buffer in the TLS which is processed either when its full, or on local synchronization events. This limits the number of *clean-calls* which significantly reduces the overhead.

To further avoid the overhead of the instrumentation itself in phases where the detector is disabled, we add a short-circuit for this case. This checks a flag in the TLS and skips the access entry related instrumentation. Hence, less instrumentation code is executed per memory referencing instruction.

Additionally, we try to limit the state modifications as much as possible, as everything that is going to be changed by the instrumentation has to be preserved before and restored afterwards. Hence, we use the Load Effective Address (LEA) instruction for calculations as this does not alter the condition codes such as `CF` and `ZF`. The instrumentation presented in Listing 4.9 only requires three registers and does not modify any `CFLAGS`. We let DynamoRIO dynamically select two registers on each instrumented section, so that dead (unused) registers can be used. This further reduces the cost of preserving the application state. However, we still have to load the TLS field and read the detector state, which adds two additional

memory accesses. These cannot be avoided and limit the minimal slowdown we can achieve on instrumented code. For even less performance impact, scoping has to be used. We further discuss this in Chapter 6.

Listing 4.9: Inline instrumentation which is added to every memory referencing instruction of the application.

```
%get_mem_addr(reg1)
%get_tls_field(reg3)

%detect_or_not # logic depends on mode

mov reg2, [reg3 + offs(bufptr)]      # load buffer pointer
mov [reg2 + offs(write)], (write)    # 1 if write, 0 otherwise
mov [reg2 + offs(addr)], reg1        # target mem addr
mov [reg2 + offs(size)], (size)      # size of access
mov [reg2 + offs(pc)], (pc)          # instruction pointer
lea reg2, [reg2 + sizeof(mem_ref_t)] # increment buffer ptr
mov [reg3 + offs(bufptr)], reg2      # update buffer ptr
mov reg1, [reg3 + offs(bufend)]      # load buffer end
lea reg2, [reg1 + offs(bufptr)]      # 0 if buffer is full
jecxz pre_flush
jmp restore

.pre_flush        # jmp into CC and clean-call
mov reg2, restore # load return addr into RCX
jmp .cc_flush     # jump into cc and flush

.restore          # restore app state
%dr restore
```

4.3.4 Synchronization Interception

In addition to the memory references, we have to detect functions involved in synchronization. This is necessary as invocations of these functions define the synchronization pattern of the application under test, which is used to identify data races (see Chapter 3.1 and Figure 3.1). These functions are wrapped with a function that determines the parameters (e.g. address of a mutex) and the return value. This is done using DynamoRIO's function wrapping extension by registering a pre- and a post- callback for a function at a given address in the "module load" event. When a fragment containing this address is loaded into the code cache, the callbacks are added.

Before any synchronization event is fed into the detector, the memory access buffer of the calling thread is processed. We call that "local-synchronization", as only the buffer of the calling thread is affected. Serializing the accesses is then in the competence of the detector. For testing purposes we also provide a mode where "global-synchronization" is used. This means, before writing a memory reference into the buffer, each thread checks if a (global) synchronization event is pending. If that is the case, all buffers are processed prior to the

local synchronization event. With Happened Before or lockset based detectors this strict synchronization is not necessary (see reasoning in Section 4.3.3).

4.4 Efficient Callstack Tracking

When a race is detected we are interested in callstacks for both involved memory accesses. This is essential for the user of the tool to understand the program flow and finally fix the race. With that information a manual inspection of the traversed code locations is possible to finally annotate or add the missing synchronization.

Ideally we could reconstruct the callstack of the second access just by using the current instruction pointer and CPU context. However, this brings us two problems: first, this only works for the second access, as the CPU context of the first access is not available anymore. Second, reconstructing callstacks on a x64 Windows is a tricky task. Normally debug helper dlls are used, but as the target application code is executed from the code cache, the observed CPU context does not fully correspond to the application context. This especially applies to the *RIP* register[Bru04, p.69]. We refer the interested reader to [Ots+18] for in-depth details on callstack walking and post-mortem analysis on x86/64 based Windows systems.

Finally, the goal is to either store a callstack per memory access or to be able to reconstruct it based on the IP after a race occurred.

Shadow Callstack As described previously, a fast and precise determination of callstacks is complicated in this setting. Due to these limitations we decided to use a shadow callstack. This means, we instrument every `call` and `ret` (return) instruction. Technically this is implemented by inserting a *clean-call* prior to the control transfer instruction which obtains the target location from the corresponding CPU registers. For calls, we track the address of the call instruction and append it to a thread-local buffer (shadow-stack). When a return happens, we pop addresses from the shadow-stack until we find one that matches the address of the call. This "climbing" is necessary as optimized programs generally directly return to a higher callee if multiple return instructions arrange successively.

Internally the instrumentation is implemented as a *clean-call*, which gives us the possibility to add additional logic. We use that, e.g. to change the detector state and to update the code cache as described more precisely in Section 6.1.3.

Relation to Memory References As stated above, we need a callstack for each memory reference to precisely locate a race. With the shadow-stack we are able to get this information, but it remains only valid for the function which is currently executing. This is visualized in Figure 4.2.

We call this sequence of memory references between two calls a `f-segment`, similar to the `segment` defined in Section 1.4. To limit the overhead of the memory tracing, only the

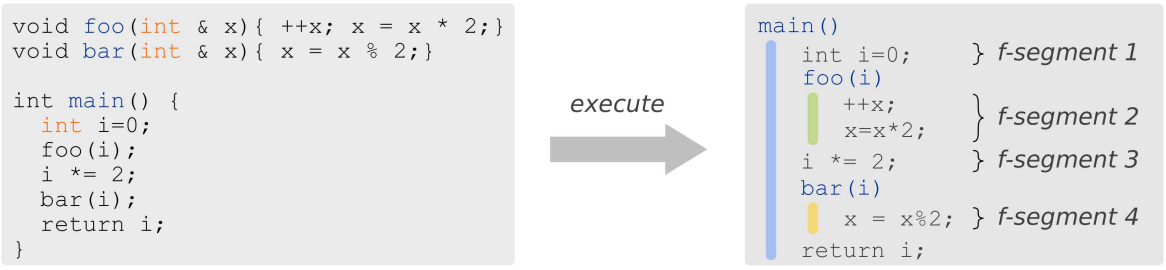


Figure 4.2: Visualization of the call-graph (colored bars) and a partitioning into `f-segments` for a sample C++ program.

current IP is stored for each memory-reference (instead of the callstack). Hence, we have to enrich each memory-reference entry with the callstack of the current `f-segment` when feeding them into the detector. This has to be done after the last instruction of a `f-segment`, as the shadow stack changes immediately after. Hence, we process the buffer containing the memory references in the `call` and `ret` instrumentation directly before changing the shadow stack. As calls occur regularly the discrepancy between the actual time memory was accessed and the time it is fed into the detector remains low. Additionally, we avoid overhead by re using *clean-calls* instead of adding new ones.

Issues and Improvements For highly compiler-optimized programs (e.g compiled with the `-O3` flag), the shadow stack provides only an approximation of what the user would expect according to the program sources. While the call hierarchy of the executed code is observed correctly, compilers heavily use inlining to avoid the overhead of calls. With inlining, the body of a function is directly copied into the calling function, instead of using a call. However, we found that this is normally no problem as the most interesting IP is the one which finally lead to the race. This is always correct, as it is determined using the memory reference instrumentation. Conceptional this becomes clear as this IP is inside the body of the calling function and hence is not a call itself.

In future iterations of DRace we plan to refine the second callstack using a debugger. By that, we can reduce the depth of the shadow stack while we are still able to print a precise callstack for the second access of a race. There is also ongoing work in DynamoRIO regarding callstack walking.

4.5 Symbol Lookup

With symbol lookup we refer to the mapping of instruction pointers to function names and locations in the source code. This mapping is needed in both directions, which means there must be also a map from function names to the instruction pointers. To intercept and wrap the synchronization functions we have to determine the corresponding function calls in the

Windows API DLLs by querying the function names. On the other hand, when a race occurs the instruction pointers of the callstack entries have to be converted into a function and possibly a source file, line and offset.

We do that using the `drsyms` extensions of DynamoRIO, which internally uses the *dbgeng.dll*[Mic17] to perform the lookup. If the functions to be looked up are exported, no further information is needed. This is the case for the Windows Sync-API functions[Mic18b], which provide POSIX synchronization mechanisms. Hence, we do not need any debug information to detect data races on accesses which are not properly synchronized according to these functions.

However, for helpful information regarding races on the target application, debug information is needed. If the debug symbols also contain line information, the involved instruction pointers are symbolized to a filename, line and offset. Additionally, debug information is necessary to wrap non exported functions which are common in C++ header only libraries.

Normally the symbol lookup is performed when a module is loaded. After the lookup and wrapping, the symbol information is unloaded to limit the memory overhead. However, for data races, this information has to be re-loaded to symbolize the callstack. As this might be time consuming for a large amount of races, we provide a mode where the symbol lookup is delayed until the shutdown of the target application. This avoids overhead during runtime. For managed code however, the symbol lookup has to be performed immediately. The situation on Dotnet code is described more precisely in Section 5.2.

4.6 Extensible Logic

One of the goals of this thesis is to provide ways to extend the instrumentation and detector logic. Thereto, parts of the instrumentation logic are not hard-coded into the client, but configured using a config file. This includes the function names (symbols) which implement a specific synchronization pattern. DRace then looks for these names in the target application and wraps them accordingly. We tested this approach successfully on a Qt5 application, where we started with default settings and iterative configured the Qt synchronization patterns.

To limit overhead or to exclude known-to-be race-free parts, functions, modules or whole file paths can be excluded from the analysis. For excluded modules or module paths, the memory tracking instrumentation is then not added. However, some instrumentation like the shadow-stack (see Section 4.4) has to be added anyways as otherwise the reported races are imprecise. We found that this is no limitation in practice, as the constant overhead is low. For detailed considerations regarding this topic, see Chapter 6.

Additionally, more advanced logic can be implemented by extending the corresponding modules of DRace. The source code of DRace is publicly available on GitHub.[1]

[1] https://github.com/siemens/drace

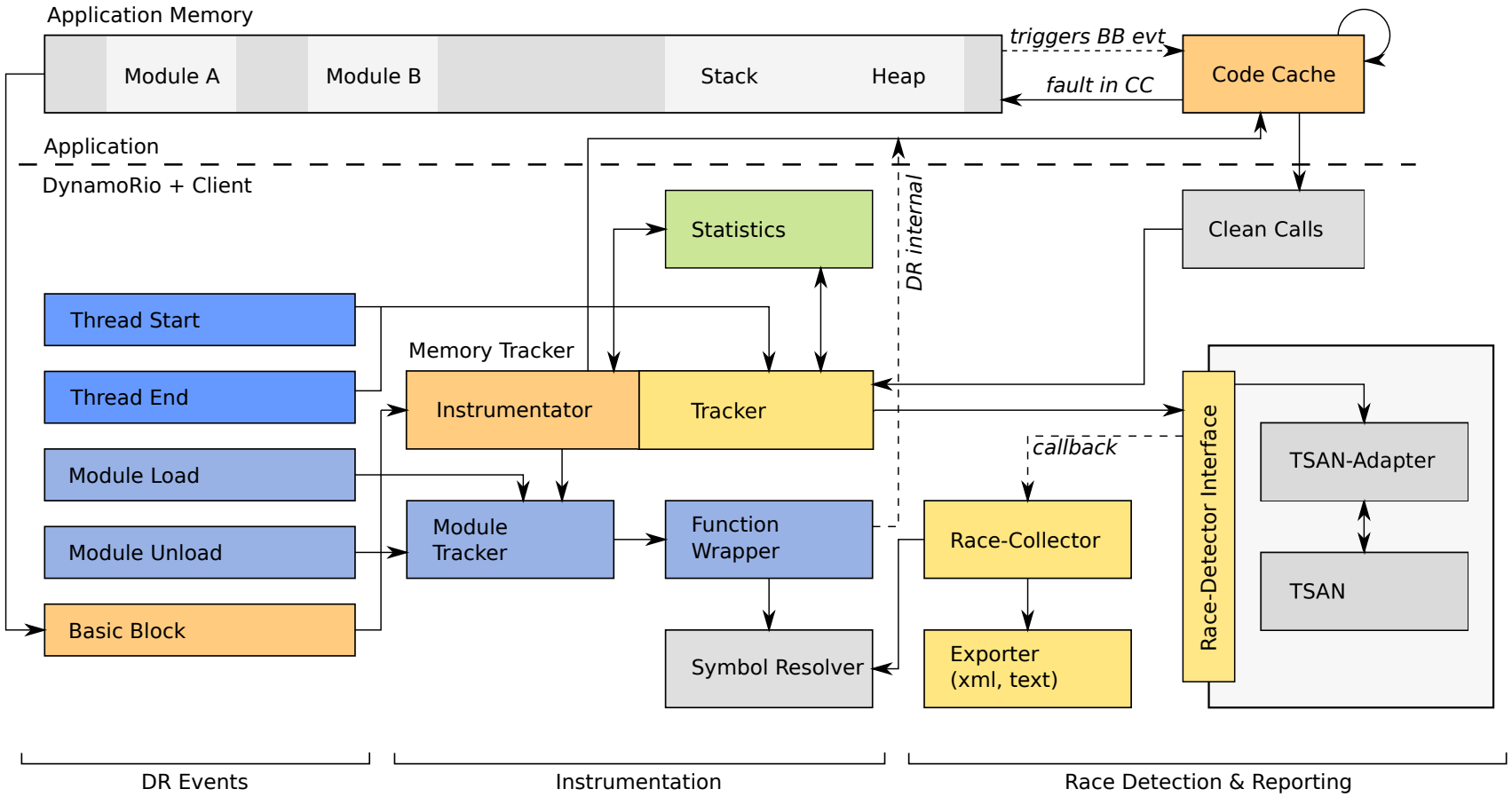


Figure 4.3: **Internal architecture of DRace**: The framework is purely event driven by events from both DynamoRIO and *clean-calls* from the application instrumentation.

5 Managed Code

In this chapter we show a proof of concept for data race detection on hybrid code consisting of native and managed parts. With managed code we refer to machine code that is executed under the management of a CLR virtual machine, whereby we focus on the Dotnet CoreCLR implementation. In this setting we target applications containing managed parts that are written in the C# programming language, which is based on the Dotnet framework.

We show that data races happening in managed code parts are detectable by analyzing the raw memory operations executed by the CPU. For this purpose, the memory referencing instructions in just-in-time compiled fragments are instrumented exactly identical to the ones in native code. While we use a uniform detection logic on both managed and native parts, managed parts require a different architecture to perform the symbol lookup and to locate synchronization contracts.

5.1 Intermediate Language

The Common Intermediate Language (CIL) is a low-level programming language defined by the Common Language Infrastructure (CLI) specification and used by the Dotnet based programming languages. In earlier versions of Dotnet it was called Microsoft Intermediate Language (MSIL) but after the standardization of the CLI this name was dropped.

During the compilation of a managed program, the source code is translated into an intermediate language rather into processor-specific (native) object code. This intermediate bytecode is then executable in any environment providing a runtime (also called VM) for this language. At this juncture, the JIT compiler compiles the bytecode into native machine code directly before execution. This native code is stored on the heap to be executed by the CPU by jumping to an address in this memory block.

One advantage of this approach is that the same program can be executed on both Windows and Linux based OSs. While it is possible to directly write programs in the intermediate language, this is not common due to the low level of the assembler like instructions.

Using programs like “ildasm.exe”, “ilasm.exe” or “IlSpy” it is possible to modify the CIL of executables and libraries before execution. These tools decompile the CIL bytecode into a human-readable representation of CIL and re-compile it after the modifications have been applied. This makes it possible to create instrumented copies of the application and its libraries.

Recent versions of the Dotnet framework add the option to pre-compile IL parts for a specific target architecture using a Native Image Generator (NGen). This eliminates the JIT overhead which reduces the startup time of the application. However, this increases the complexity when modifying the CIL, as both the compiled and the IL parts have to be changed. As recent Dotnet system libraries already use this approach, DRace has to support these cases as well. At time of this thesis we do not know of a tool that supports IL modifications on modules containing NGen'd parts.

5.2 Managed Functions

Function wrapping is essential to detect managed synchronization mechanisms like mutexes. Experiments and theoretical examinations have shown that it is not sufficient to track the related system calls, as optimized implementations of C# use a mixture of system and user level synchronization techniques. Consider a fast lock implementation as an example: at first a user-level Spinlock is used. If this spinlock cannot be acquired in a few cycles, a system lock is used as fallback. This observation requires us to directly detect synchronization in the managed part of the code, as otherwise synchronization contracts are missed.

There are various approaches to achieve this:

1. Modification of the CIL
2. Replacing the Dotnet `System.Threading` DLL with an instrumented version
3. Using the `CLRProfiling` infrastructure
4. Instrument JITed fragments directly using DynamoRIO

These approaches are evaluated regarding the following aspects:

- **Compatibility**: Are modifications necessary to support most Dotnet versions. Is it likely that this breaks with future Dotnet implementations (higher is better)
- **Performance**: Expected slowdown of the application (higher performance is better)
- **Technical Complexity**: Amount and complexity of the implementation (lower is better)
- **Documentation Available**: How well is this approach documented. This includes API documentation, reference implementations, blog posts and entries on Stackoverflow.

A tabular comparison of the relevant aspects is given in Table 5.1.

Method	Compatibility	Performance	Complexity	Doc. Avail.
(1) CIL Modification	high	high	impossible	-
(2) Threading Mod. Repl.	low	high	medium	- -
(3) CLR Profiling Infra.	medium	low	medium-high	+
(4) JIT-Fragment Mod.	high	medium	low	++

Table 5.1: Comparison of a set of approaches to intercept Dotnet internal synchronization mechanisms. All ratings are based on theoretical considerations

After an initial evaluation of all described approaches, the only one that proved to be realizable was the instrumentation of JITed fragments using DynamoRIO. This method is presented in Section 5.2.4. For the sake of completeness and as a starting point for future work, we briefly discuss the other approaches as well in the following sections.

5.2.1 CIL Modification

Instead of instrumenting the JITed machine code, the basic concept of this strategy is to add the instrumentation at CIL level. For synchronization primitives, it is sufficient to add a pre and a post function call into a non-managed library. The calls to these functions are then intercepted and fed into the race-detector.

With common Dotnet instrumentation techniques like `Mono.Cecil` the approach to modify CIL is the following: At first the CIL is loaded from the DLL, then the tool analyzes it and adds the instrumentation code. In a final step, the modified intermediate language is written to a new DLL.

However, this is has shown to be inappropriate as the managed DLL is already loaded into the programs address space after the module load event has fired. Hence, no further modification of the DLL is possible. Additionally, we did not find a solution to instrument modules that contain NGen pre-compiled parts along with the IL parts.

5.2.2 C# Threading Module Replacement

All C# synchronization routines are at least wrapped with a tiny C# layer. The idea is to replace the wrapper with a custom version that implements the same functionality but additionally informs the detector backend about this synchronization event.

To be compatible with different Dotnet versions, we tried to generate this wrapper based on an existing version using the `Mono.Cecil` framework. However, this approach was dropped after empirical tests, as various issues showed up: The modules are often loaded using absolute paths, which makes interception tricky. Modern C# applications are often shipped with a self-contained Dotnet environment, which makes it hard to pre-build modified DLLs for all Dotnet versions. Furthermore we did not find a solution to modify the CIL in DLLs con-

sisting of managed and native parts. For the `System.Private.CoreLib.dll` which provides parts of the synchronization implementation, this is unfortunately the case.

5.2.3 CLR Profiling Infrastructure

A profiler is a tool that monitors the execution of an application during runtime[Wen17]. The CLR profiling infrastructure specifies an adapter between messages from the CLR and the Windows profiling API[Lan07b]. By implementing the interface, the user is able to select events he is interested in and to register a callback for each event.

Unfortunately, the interface is not well documented and there are only rare reference implementations available. To use it, dozens of functions have to be implemented or mocked before testing. Additionally, it remains unclear if it is compatible with DynamoRIO as it might rely on hardware states which are not fully transparent to the API (see Section 4.3.2). Due to time constraints of this thesis we decided to choose a different approach.

5.2.4 JIT-Fragment Modification

The general idea of this approach is to instrument the JITed machine code similar to the instrumentation described in Section 4.3.4 using DynamoRIO. The main challenge is posed by locating synchronization contracts in these fragments. Thereto we need a mapping between managed symbol names and IPs in the JITed code fragments.

Managed Code Debugging We observed that symbol resolution of hybrid applications is supported by the "WinDbg" debugger using the "sos.dll" debugger extension. This extension uses helper DLLs provided by the Dotnet runtime implementation to map just-in-time compiled addresses to the CIL and finally to a location in the source files. As the debugger extension only provides a textual API which is intended to be used from the debugger command line, we use the interface of the Dotnet runtime helper DLLs directly. For the CoreCLR implementation, this is "mscordaccore.dll", for CLR it is "mscordacwks.dll". These libraries implement functions and macros to access the internal Data Access Component (DAC) structures which describe how managed code is mapped to physical (virtual) addresses [Lan07a]. Finally, the DAC is available to an attached debugger that implements the "ICorDebug" interface. An overview of this architecture is provided in Figure 5.2.

The DAC helper dynamic library cannot be loaded into the DRace process as it is incompatible with DynamoRIO's private loader. Hence we use a second process called Managed Symbol Resolver (MSR) for this task. The architecture of this solution is shown in Figure 5.1. The MSR communicates with DRace using shared memory and attaches to the process of the target application using the Windows debugging API. Here it is important to attach in the non-invasive mode as otherwise the interrupt in the target application clashes with DynamoRIO and results in a crash. In non-invasive mode, the debugger suspends all

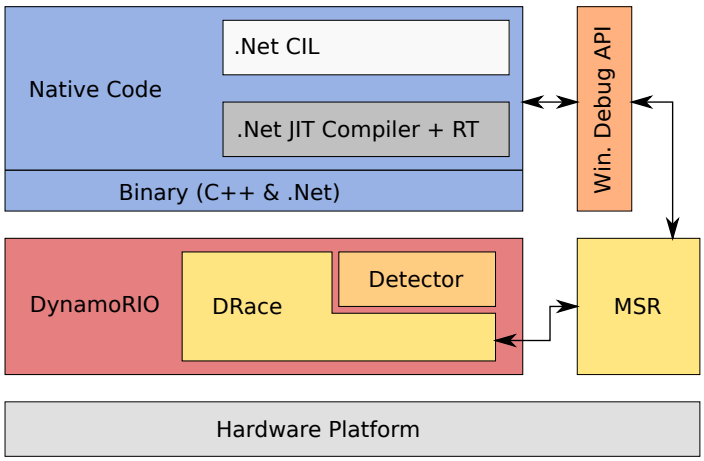


Figure 5.1: Layered architecture showing a hybrid application running under DRace. A second process (MSR) is used for symbol lookup in the managed code parts. Both processes communicate using shared memory.

of the target's threads and gains access to its registers and memory, but it does not modify the execution behavior apart from that.

External Symbol Lookup Symbol lookup requests of DRace are forwarded to the MSR and the result is returned to DRace again. All code manipulations including function wrapping and callstack tracing are performed in DRace. Using the MSR, we are able to download missing symbols from a Microsoft Symbol Server. This is not possible for a DynamoRIO client, but as the MSR is executed as a standalone process these limitations do not apply. When DRace detects that a managed module is loaded, it notifies the MSR which then issues a download of the debug information for this library. The automated symbol download is beneficial as Dotnet versions require perfectly matching debug information.

To locate the native address of a synchronization primitive, we query the corresponding symbol name using the MSR. After that we wrap the returned address with our synchronization handling routines and process it just like native synchronization procedures (see Section 4.3.4). Data races also occur on native (virtual) memory addresses. When a race is detected we determine the symbol name of each callstack entry from the corresponding instruction pointer. If the IP belongs to a managed module, we perform the symbolization using the MSR and pass the result back to DRace. There, managed and unmanaged callstack entries are merged and presented uniformly.

5.3 Synchronization Primitives

All synchronization primitives are declared in `System.Private.CoreLib.dll`. However, some methods (e.g. `System.Threading.Monitor`) are implemented using native code.

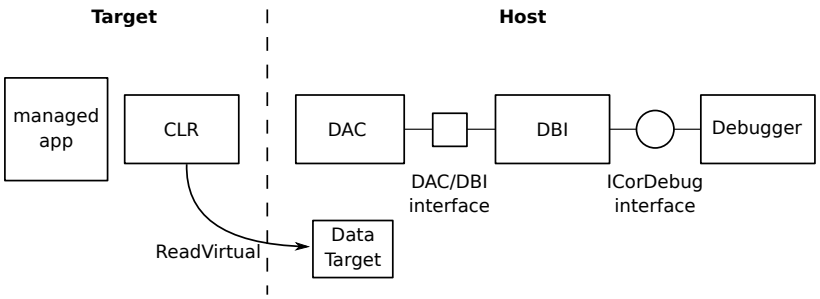


Figure 5.2: Scenario where an external debugger attaches to the managed application using the ICorDebug interface and the DAC structures. [Lan07a]

For native functions we need debug information and rely on the *drsyms* infrastructure of DynamoRIO. If the debug information is not available, we trigger a download from the Microsoft Symbol Server, using the MSR. For managed symbols, DAC helper support is necessary. Hence each lookup request is passed to the MSR and all matching IPs are then returned to DRace. The wrapping of the functions is done in DRace using DynamoRIO's *drwrap* infrastructure.

Dotnet Monitors An important and special Dotnet synchronization technique is the `System.Threading.Monitor` class. Internally, locks on an object (`lock(x){...}`) are forwarded to the monitor. Mutual exclusion is then initiated (internally) by `Monitor::Enter` and left with `Monitor::Exit`. Here, `Enter` semantically corresponds to the acquisition and `Exit` to the release of a mutex.

The (managed) function heads of these methods are decorated with the attribute `[MethodImplAttribute(MethodImplOptions.InternalCall)]`. When viewing the `System.Threading.Monitor::Exit` function in ILSpy, the disassembly looks as follows:

Listing 5.1: CIL disassembly of a Dotnet Monitor Enter.

```
.method public hidebysig static
void Exit (object obj) cil managed internalcall
{
} // end of method Monitor::Exit
```

The `internalcall` attribute tells the JIT to replace this occurrence by a native function call. The pointer to this function is stored in an internal virtual method table in the JIT and depends on the system architecture. To intercept a call to a function of this kind, we have to intercept the native function call. However, the exact symbol names heavily change between Dotnet versions. For the Dotnet CoreCLR implementation the names are specified in the `ecallist.h` header[1]. At time of this thesis, we specify the symbol names manually in DRace' configuration file for the last Dotnet CoreCLR implementations. In future iterations

[1] `https://github.com/dotnet/coreclr/blob/a9c90ffd84e987a1793a1ed5c5ad7d89b27d493a/src/vm/ecalllist.h#L1824-L1833`

we plan to automate this process by reading the names directly form the corresponding header file.

5.4 Separation of Native and Managed Code

For the data race detection itself no separation of managed and unmanaged code parts are necessary. However, to locate and wrap managed synchronization procedures as well as to symbolize managed frames in a callstack, we have to apply different logic on managed and native modules. Before managed code can be executed the just in time compiler compiles the intermediate language into machine code. The machine code is then either stored inside the address range of a managed module, or on the heap. In the latter case, the JITed instruction pointers are not in the address range of any module.

In the module load event we determine if the current module is managed by looking at the `COM_ENTRY` value in the data directory header. If the value is zero the module contains only native code. Otherwise it is a managed module [Mic18a]. The full header walking is shown in Listing 5.2.

When a race occurs, each callstack entry is symbolized. Here we check if it is inside any module and if it is a native module. In this case the symbol lookup is performed using DynamoRIO. In all other cases the instruction pointer is managed and we perform the symbol lookup using the MSR.

Listing 5.2: Header walking to determine if a module is managed.

```
#include <Windows.h>
IMAGE_DOS_HEADER       pidh = (PIMAGE_DOS_HEADER)module->start;
PIMAGE_NT_HEADERS      pinh = (PIMAGE_NT_HEADERS)((BYTE*)pidh + pidh->e_lfanew);
PIMAGE_OPTIONAL_HEADER pioh = (PIMAGE_OPTIONAL_HEADER)&pinh->OptionalHeader;
// clr != 0 if managed
DWORD clr = pioh->DataDirectory[IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR].VirtualAddress;
```

5.5 Evaluation

As a proof of concept of our uniform analysis approach, we analyze a managed demo application based on the Dotnet CoreCLR implementation. The application is used to verify the detection of synchronization mechanisms as well as our assumption that a tracking of physical / logical memory accesses is sufficient.

5.5.1 Demo Application

The demo application acts as a test to verify that user-level synchronization procedures are processed correctly. Additionally, we test the symbol lookup of the callstacks in case of a data race.

To locate the Dotnet internal synchronization procedures, debug symbols containing source information are necessary. For Dotnet internal modules this information is automatically downloaded from a Microsoft Symbol Server using the MSR. However, to symbolize the callstacks correctly this is not sufficient as debug information on the target application itself is required. Otherwise stack frames which belong to these sections cannot be resolved. Unfortunately, the default setting of Microsoft Visual Studio is to not include this kind of debug information in the PDB files.

The logic of the demo application is to concurrently increment a shared counter using n threads. Here, we test three settings:

1. racy increment
2. increment guarded by a `lock()`
3. increment guarded by a mutex

The source code of the application is attached in Appendix 2. In setting (1) we expect DRace to find a race. In the two other settings we expect the code to be race-free. The main difference between setting (2) and (3) is that the first one implements user-level synchronization. The latter always uses an OS lock, which is already tracked by the native instrumentation.

5.5.2 Results

Indeed the race was correctly detected in setting (1) and no races were detected in the two other settings. The main slowdown of the application happened during the startup of the Dotnet runtime. Especially the download of the symbols of the Dotnet internal modules takes significant time on the first execution. This exact duration mainly depends on the throughput of the Internet connection and was approximately a minute in our tests. For later executions, the symbols are already in the local cache, which avoids this delay.

To get precise callstacks inlining of the C# code has to be disabled currently. This is due to the shadow stack which is used instead of stack walking which could possibly use debug information to circumvent this limitation. Listing 5.3 shows the race as it is reported by DRace. At the time of this thesis DRace is not capable of mapping a managed instruction pointer to a source file and line. Currently only the module and function name can be resolved.

Listing 5.3: Race reported on a concurrent increment in managed code.

```
Access 0 tid: 20696 read to/from 0000000000034D40 with size 1. Stack(Size 10)Type: -1
    ↪  :
Block not on heap (anymore)
#0 PC 00007FFBB74F23DF (dynamic code)
        Module TestCS-win64-racy-exe\TestCS.dll - MultithreadingApplication.
            ↪ ThreadCreationProgram.IncByOne(MultithreadingApplication.param)

#1 PC 00007FFBB74F0078 (dynamic code)
        Module JIT - MultithreadingApplication.ThreadCreationProgram.IncByOne(
            ↪ MultithreadingApplication.param)

[...]
#9 PC 00007FFC16FBDF84 (rel: 00000000000FDF84)
        Module coreclr.dll - ThreadNative::KickOffThread_Worker
        from 00007FFC16EC0000 to 00007FFC1740B000
        File e:\a\_work\308\s\src\vm\comsynchronizable.cpp:257 + 0
Access 1 tid: 10496 write to/from 0000000000034D40 with size 1. Stack(Size 1)Type: -1
    ↪  :
[...]
```

6 Performance Optimization

From a theoretical perspective, the execution speed of a physical machine must neither change the correctness, nor the behavior of an application running on it. In theoretical computer science, the physical machine which executes a program is abstracted by a Turing machine (TM). The program is then defined as the strip of tape which is the input of the TM. It is further possible to simulate a TM with its input by using a second TM. Conceptional this analogy is very similar to what we present in this thesis: we take a program which can be run on a target machine and simulate it on another machine. As the TM has no sense of physical time, but only logical steps, the input $\Leftrightarrow$ output relation of the program (strip) is not changed.

However, this only holds in theory, as in practice resources are limited. This especially applies to the time it takes to perform this simulation, which finally bounds how many logical steps can be executed. The simulation approach only works for applications which can be fully simulated including the surrounding environment and hence do not rely on any external state. This is a different field of application which is not focused by this thesis.

Instead of simulating the program, we modify (instrument) it and execute it natively. This is done in a manner that these changes are hidden from the perspective of the target application. In practice, large applications are not fully self-contained, but depend on external components. This can be e.g. calls to external APIs which have to be processed within an externally defined time span. Additionally, the development cycles consisting of running DRace and fixing races should be as tight as possible. Hence, we have to reduce the overhead of DRace to finally meet these external constraints.

As most of the overhead is induced by the detector, we have to reduce the amount of data it has to process. This can be done either by sampling, or by excluding application parts (scoping) from the analysis. Given that, less memory references have to be processed and program execution is less retarded. This approach is especially interesting for interactive applications (e.g. GUIs) and long-running procedures. While sampling excludes elements at a regular interval, scoping completely excludes regions of the target application. Hence we will find data races in areas where sampling is used with a given probability. In contrast to that, we will never find data races in areas which are not in the analysis scope.

In this chapter we cover various sampling and scoping strategies. These are evaluated regarding their performance, universal suitability and impacts on the quality of the race detection.

6.1 Sampling

With static sampling we mean to process only a small subset of memory references by sampling approximately each T'th memory access. This assumes that the parts of the application which should be analyzed are executed multiple times. Both server processes and GUI applications mostly consist of three phases: A startup phase, a steady state phase where the main logic happens and a shutdown phase. The steady state phase is normally implemented as a loop, which means that the same instructions are executed over and over again. This makes it likely that a race is detected, even if only a small part of memory references are analyzed. In contrast, races during the startup and shutdown phase are unlikely to be detected with sampling.

We expect this approach to work well only if the following conditions are met:

- functions are executed multiple times
- the application has a `steady-state`
- the application is executed long enough

6.1.1 Probability Estimation

While sampling is useful to limit the overhead, there is a loss in the detection accuracy. With accuracy we mean the number of missed races, which maps to the amount of false negatives. Here we consider a pure happens-before based detector, but we expect the calculations to also apply to hybrid detectors.

For being able to detect a race, we have to capture the two non-synchronized accesses. The probability to sample both accesses in a series of x events is calculated according to the urn problem[1]. The fundamental question we ask is: given a sampling period T and a stream of memory-referencing instructions $[I_1, I_2, \ldots, I_t]$, what is the probability to catch both references at least once when each racy access occurs m and n times.

Thereto we first compute the total number of samples as $s = \frac{t}{T}$, where t is the total number of instructions and T is the sampling period. The probability p for catching both memory references at least once is then:

$$p = 1 - \frac{\mathcal{A}_{\neg m} + \mathcal{A}_{\neg_n} - \mathcal{A}_{\neg(m \cap n)}}{\mathcal{T}} = 1 - \frac{\binom{t}{s-m} + \binom{t}{s-n} - \binom{t}{s-m-n}}{\binom{t}{s}} \tag{6.1}$$

Here, $\mathcal{A}$ denotes appropriate selections and $\mathcal{T}$ the total number of combinations to sample s events in a sequence of length t. For $\mathcal{A}_{\neg m}$, we consider all combinations where we miss the

[1] This estimation is based on [She+11, p. 405], but fixes the calculations by correctly labeling the variables s and t, as well as fixing the binomial coefficient.

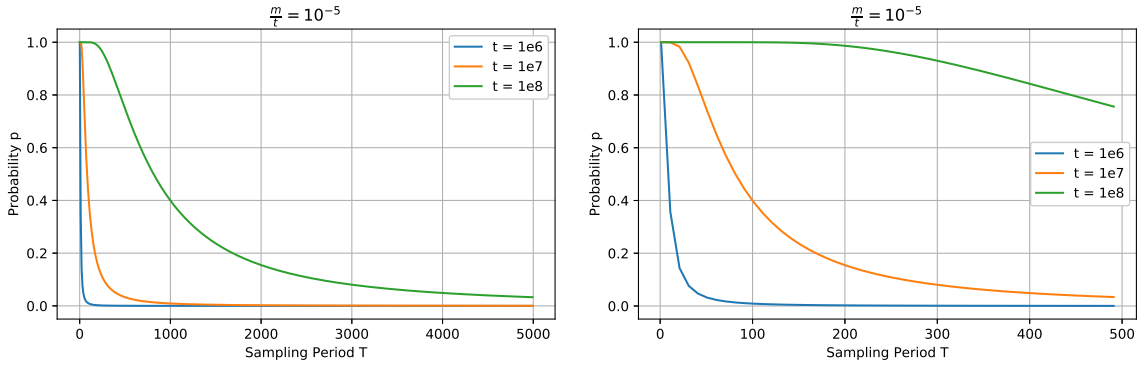


Figure 6.1: Probability to find a data race with various sampling periods T and sequence length t, based on theoretical considerations. We assume that $\frac{m}{t} = 10^{-5}$ of the total executed (memory-referencing) instructions are racy.

first access, $\mathcal{A}_{\neg n}$ where we miss the second access and $\mathcal{A}_{\neg(m \cap n)}$ where we miss both accesses (these are counted twice otherwise).

$$p \approx 1 - \left(1 - \frac{m}{t}\right)^s - \left(1 - \frac{n}{t}\right)^s + \left(1 - \frac{m}{t} - \frac{n}{t}\right)^s \tag{6.2}$$

The approximation assumes that after an instruction is sampled, it is returned to the pool and can be sampled again. Additionally, we assume that instructions are sampled independently. If $t >> m$ and $t >> n$ holds, this does not significantly affect the calculation as the Equation in 6.1 converges against the Equation in 6.2 (Stirling's approximation).

In practice we do not use a fixed period but a probabilistic approach (see Section 6.1.2). This mitigates the issue that instructions might not be sampled independently.

As both $\frac{m}{t}$ and $\frac{n}{t}$ are properties of the program, we must increase s to increase the probability of detecting a race. This can be done by either reducing the sampling period T or running the program longer, which increases s.

As we do not directly target extremely long-running applications like [She+11] did, we empirically found the following parameters to provide good results: $\{t = 10,000,000, T = 25, \frac{m}{t} = \frac{n}{t} = 10^{-5}\}$. Hence we get $s = 400,000$ samples and compute the final probability to find a race to be 96%. In contrast to [She+11], we found that races are less likely then $\frac{m}{t} = 10^{-4}$. In our opinion this is too optimistic for non-service applications. Hence, we recommend using $\frac{m}{t} = \frac{n}{t} = 10^{-5}$. A comparison of the race detection probability and the sampling rate is given in Figure 6.1.

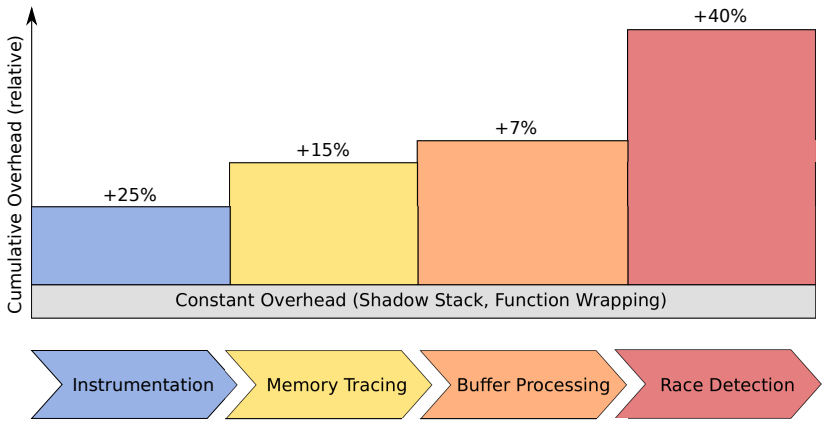


Figure 6.2: Each step of the processing pipeline adds overhead to the execution. As the actual overhead heavily depends on the target application only estimates are given. The constant part of the overhead cannot be avoided.

6.1.2 Sampling without systematic error

Consider the processing pipeline showing all steps necessary to finally analyze a memory reference:

1. **Instrumentation** Code blow-up by adding instrumentation assembly
2. **Memory Tracing** Instrumentation code writes metadata into a buffer on each memory access
3. **Buffer Processing** The buffer is processed, filtered and passed to the detector
4. **Analysis** Finally, the memory references are analyzed in the detector

We have to sample as early as possible, but without a systematic error. Sampling in instrumentation provides maximum performance boost, but the number of missed references is unpredictable.

For later stages than *Memory Tracing*, large amounts of the total overhead did already happen (see Figure 6.2). So it is best to implement sampling before the accesses are actually imposed. A visualization of both sampling techniques and scoping is provided in Figure 6.3.

6.1.3 Block Based Sampling

As our instrumentation already provides a short circuit for a disabled detector, this could be used to skip references. However, we are then limited to block-based sampling, as the state-flag does not impose a counter.

Given that the sampling is implemented using the detector state, we have to minimize the overhead of state-changes. For good analysis capabilities, we prefer a uniform distributed

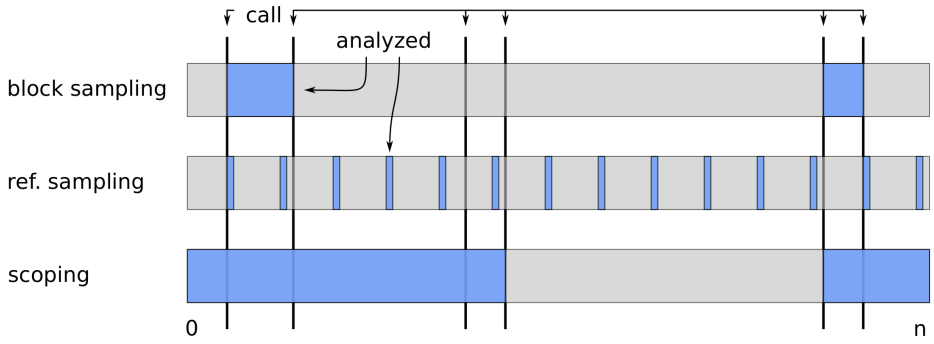


Figure 6.3: Differences between two sampling techniques and scoping visualized on a sequence of n memory references. The blue ranges are analyzed.

sampling with a minimal variance. However, before changing the state, other state-related characteristics have to be checked, so this has to be done in a *clean-call*.

As the *clean-calls* are relatively expensive (compared to inline instrumentation), it is best to use an already existing one. The call should be executed after roughly the same number of references each time, to keep the variance low. A suitable *clean-call* is the call event of the shadow stack, as empirical analysis showed that this happens after 8 to 16 references in average.

We also experimented with other techniques to change the state, but the approach described above has turned out to have both minimal overhead and no interference with the target application. A promising idea was to use a shadow thread for the switching, as this avoids systematic errors. With the approach described above, the granularity is always a whole `f-segment` (sequence of instructions between two consecutive calls). However, the overhead of an additional thread including atomic instructions to modify the switch has proven to be way to high. This is also due to the frequent wakeups, as the detector has to be toggled in very short intervals.

Fast estimator for sampling decision

As the state should be changed after each bunch of references, the Random Number Generator (RNG) must not create a bottleneck. However, to avoid systematic errors, the quality of the random numbers has to be at a reasonable level. Hence, some trivial approaches like using a counter and the reminder of a division are inappropriate. An example of this case would be a loop containing 100 memory referencing instructions and a sampling rate of $T = 5$. Without adding a random delta to each period, the sampling would always select the same 20 memory references. To further avoid this systematic error, we recommend choosing a prime as period length T.

In our implementation we use a Mersenne-Twister Engine (MT) and a uniform integer distribution to calculate the next period length. The length is calculated to be $T \pm 10\%$

to add some variance and break circular loops. During execution we just decrement the sampling counter until it becomes 0. Then the next period length is calculated. This further reduces the overhead per skipped reference, as only a single decrement and one comparison is necessary to make the decision to sample or skip the next block. For small sampling periods we re-calculate the period length only every few iterations to reduce the overhead of the MT. Additionally, the period length has to be an integer value, so for periods smaller $T = 10$, we calculate $T_{next} = T \pm 1$ to get periods with variable length.

6.1.4 Reference Based Sampling

This approach implements sampling directly on the number of memory references, instead of the number of function calls. Hence, we expect the results to closely map our theoretical considerations from Section 6.1.1. However, this comes at the cost of an additional overhead per memory reference:

While we used the detector state flag to decide if a reference is going to be sampled, this is not sufficient here. On each memory access, we have to increment our position in the sampling period. If the period length is reached, we have to sample this reference. Otherwise the reference is skipped.

This can be implemented either in the memory tracing or in the buffer processing stage. In the first case, the sampling logic is directly implemented into the inline instrumentation which is added around every memory reference. However, as the size of this instrumentation is limited by DynamoRIO this requires us to instrument with fewer instructions. Thereto we have to use assembly instructions which modify the condition flags, finally requiring us to save and restore them. This enables us to optimize away a memory access, as we combine the sampling counter and the detector state in a single 64 bit value. The instrumentation assembly code is provided in Listing 6.1. Here, we use the following assembly to decide if this reference should be processed:

Listing 6.1: Detector state and sampling logic added to every memory referencing instruction.

```
mov reg2, [reg3 + offs(ctrlblock)] # load controlblock
bt reg2, 63                        # check bit 63 (detector state)
jcc JB, restore                    # if set, short-circuit
sub [reg3 + offs(ctrlblock)], 1    # decrement sampling counter
jecxz process                      # process this ref
jmp restore                        # skip this ref

.process
mov reg2, [reg3 + offs(period)]    # load sampling period
mov [reg3 + offs(ctrlblock)], reg2 # reset sampling cntr
# ... put this reference into the buffer
```

In the second case where we sample in the buffer processing stage, all memory references are written into the buffer. This simplifies the inline instrumentation logic, but adds a significant overhead per reference. Sampling is then performed on the values in the buffer, by feeding just some access entries into the detector. There is still a reduction in the runtime overhead, as most overhead is created in the race detection stage (see Figure 6.2), but this contradicts our goals described in Section 6.1.2.

6.2 Scoping

In contrast to sampling, scoping means to just include some parts of the application in the analysis. Other parts, which are not in the scope are completely excluded. Given that, data races which are not in the scope of the analysis will never be found. However, if the scoping is applied dynamically, only high-traffic areas of the program are excluded. The key idea here is to execute all parts at least a few times, before the high-traffic parts are excluded. This gives a good tradeoff between correctness and performance.

6.2.1 Dynamic Scoping

One problem with static sampling is that we do not distinguish between high and low-traffic parts of the application. Hence, the quality of the detection is bound by the sampling factor and the requested slowdown. Mathematically spoken, detection quality and slowdown are inversely proportional.

Often the maximum slowdown of the application is limited by real-time constraints, like timeouts of calls to external resources. Hence, the sampling rate has to be chosen such that these limits are not exceeded.

Dynamic Approach

To lessen this problem, frequently executed parts of the application can be sampled with a smaller density than the rest of the application. As it is not possible to pre-determine these parts during the analysis phase, this has to be done while the application is running.

A simple approach would be to build a histogram of the observed instruction pointers and change the instrumentation on the top-k (most frequent) blocks. However, in its unoptimized version this is not possible as the tracking of the IPs adds more overhead than the instrumentation itself.

Altering the Instrumentation

Given that we have a possibly approximated histogram of frequent IPs, the next step is to change the fragments in the code-cache. We are interested in fragments which contain a frequent IP, but there is no efficient direct mapping.

As already covered in previous chapters, the instrumentation is added whenever a fresh fragment is moved to the cache. This happens when the fragment is going to be executed, or when the fragment becomes a trace. Traces are high-traffic parts of the target application, which are handled differently by DynamoRIO.

In both cases, the fragment is already in the cache, as it has already been executed. Given that, we have to locate the fragment in the cache, flush (erase) it in a safe way, and instrument it differently on the next execution.

Altering Algorithm

The algorithm to update the instrumentation consists of two parts: First we have to remove the affected fragment from the code cache. After that, on the next execution of this fragment, DynamoRIO will detect that it is not in the cache and calls the "basic-block" event. In this event, we check if the block is in the current top-k and add only our minimal instrumentation.

Algorithm 2 Schematic version of the block altering algorithm.

```
{Flushing the cache}
store current and previous top-k histogram
calculate set-difference between both histograms
issue a delayed flush (erase) of the approximated block

{Re-instrumentation}
wait for next bb execution (not in cache anymore)
if block in top-k histogram then
  insert lightweight instrumentation
else
  proceed as normally (possibly re-add full instrumentation)
end if
```

For efficiency reasons we provide a mode which just relies on DynamoRIO's internal classification of fragments. This simplifies the implementation as the altering of the cache is done in DynamoRIO internally. Besides the reduced complexity, the overhead of tracking IPs and building histograms is avoided. However, this approach allows no fine-grained tuning based on the histogram of executed IPs.

Space Efficient IP Tracking

Computational intense parts of an application have a high instruction locality most times. This is necessary to keep the next instructions in the L1 and L2 cache.

Hence, it is sufficient to track only parts of the IP's address. Our evaluation showed that a good strategy is to round the pcs down to 128 byte boundaries. This means that on a 64 bit system, the histogram is built upon 16 element blocks, aligned at 128 byte boundaries. The good empirical results are also due to the observation that most basic blocks are slightly smaller than 16 elements.

To avoid an ever-growing histogram, we use the *Lossy Counting* algorithm. This saves space and thus computation time by still giving some guarantees on the accuracy of the counters. The Lossy Counting Model is parameterized by a frequency (f) and an error threshold (e).

Here, our evaluation showed that good results can be achieved with $f = 0.02$ and $e = 0.002$. This means that the blocks survive, which had a share of at least 2%. For elements between 2% and 1.8% we might get false-positives.

Alternative Approaches: Sliding Windows While the model above retains counters for the complete history of accesses, sliding window histograms could be used as well. These have the advantage that they adapt better to changes in the working set of the running application. However, it has to be ensured that no ping-pong between two histograms occurs. This approach is conceptional prone to this case, as previous frequent addresses are not tracked anymore after the cache is updated. In its worst form, this degrades to static sampling with a huge overhead of updating the code cache.

Exclude Code from Analysis

Given that we have a histogram containing the most frequent instruction pointer prefixes, we have to find the related basic blocks to alter the instrumentation. We do that efficiently by dividing the address of the basic block by the block size (16 elements) and check if it is in the top-k entries.

There are two ways to exclude a frequent block from the processing: at instrumentation time and at execution time. The best performance improvement is achieved, if the memory related instrumentation is actually removed from the basic block. However, this requires us to remove this block from the code cache and to re-instrument it (without memory tracing code). This adds significant overhead, if the histogram often changes.

The other option is to just disable the instrumentation during the execution of this block (see "short-circuit" in Listing 6.1). While this avoids the cache altering and re-instrumentation overhead, it adds additional overhead per block as the histogram has to be checked. Additionally, each memory reference is still instrumented and the analysis part of the instrumentation is just skipped. This reduces the locality of the machine code, which finally leads to reduced L1-Cache hit rates.

We found that there is no silver bullet regarding this topic. Both strategies have their pros and cons which finally depend on the target application. Therefore we provide this as a runtime flag to let the use decide which strategy to use.

Exclude Memory from Analysis Observing the actual binary execution is a blessing and a curse at the same time: While we can analyze dynamically assembled and hybrid applications, we have to deal with non-application related artifacts. These artifacts lead to many false positives, as they are out of the scope of our synchronization tracking. Examples are writes into the virtual address space of the kernel. We see these accesses, as we intercept and observe parts of the application "beneath" the Windows-API. Hence, we drop all memory references which are outside the virtual address space of the executing process.[2]

To limit the overhead of DRace, we further support excluding stack addresses. The stack is almost always accessed local only and pays for a decent share of all memory references. As this might hide some true races, this feature is controlled by a flag. The address range of the stack of each thread is queried in the "thread-start" event using the Windows-API.

6.2.2 Static Scoping & External Control

Often an analysis of the whole program is not necessary, as critical logic might be implemented in small modules. Other use cases are iterative tests, where known-to-be race free parts are already marked. However, with classic race detectors this "scoping" is not possible as they cannot be controlled during runtime.

We support scoping on the following levels of granularity:

1. exclude module and file paths
2. exclude functions
3. exclude static-memory accesses (sometimes misleadingly called "stack" accesses)
4. toggle the detection using an external process

The most intuitive way to analyze a module of a large GUI application is to use the external control. By that, the operator starts with the detector being disabled, navigates to the interesting module and enables it. This reduces iteration cycles, as the memory intense startup phase of the program is skipped. Additionally, this does not require knowledge on the internal structure of the application.

The external control feature is implemented by using a second controller process, which communicates with DRace using shared memory. The controller also supports to modify other processing characteristics, like the sampling rate.

[2] `https://docs.microsoft.com/en-us/Windows-hardware/drivers/gettingstarted/virtual-address-spaces`

Excluding modules and paths is useful to exclude the system libraries, which are also visible to DRace. Furthermore, custom or user-level synchronization functions itself should be excluded, as they almost always create false-positive races due to concurrent accesses on the synchronization logic.

6.3 Evaluation

We evaluate both sampling and (dynamic) scoping approaches on a small application. This enables us to validate our model in a controlled environment. For a case study on real world applications, see Chapter 7.

The mini-app is implemented to measure the effects of sampling on the accuracy of the race detection. Here, we consider two scenarios: one without any synchronization and a more realistic scenario, where a lock is used but some accesses bypass the lock. The application is parameterized by a period length, a number of rounds and the mode (sync / no-sync). In each period, exactly one non-synchronized access is performed. In the synchronized setting, one additional access - which is guarded by a lock - happens. All other accesses in the period are local only, giving $O(T)$ total accesses for a period length T. The number of rounds selects how often the test is repeated.

This closely represents the model described in Section 6.1.1 by setting $T = \left(\frac{m}{t}\right)^{-1} = \frac{t}{m}$. So the period between two races is the inverse of the number of races per memory access. Hence, $\frac{m}{t} = 10^{-5}$ corresponds to $T = 10^5$. This is the scenario we evaluate here.

It is tricky to empirically estimate the probability to find a race, as this requires a precise control over the total number of memory accesses which is not possible. Instead, we measure the time until a race is found for a given sampling rate. This directly corresponds to a variable parameter t of our model. The longer the application is executed, the more memory references are processed. Additionally, we measured that the throughput in references/sec remains approximately constant. Figure 6.4 provides a visualization of this setting for various sampling rates.

We observe that for sampling periods up to 2^7 the time to find the race is approximately equal. In this case, the race is almost instantly found after the application starts. The constant overhead ($\approx 2s$) is being caused by the time it takes to analyze the basic blocks and to add the instrumentation. As the application is small, all instrumented blocks fit into the code cache. Hence, the overhead due to adding instrumentation appears only at startup time.

For larger sampling periods, we see an exponential growth in the time it takes to find the race. We expect that according to the theoretical model in Section 6.1.1, as this range is past the “elbow” of the probability graph shown in Figure 6.1. In the experiment we run the

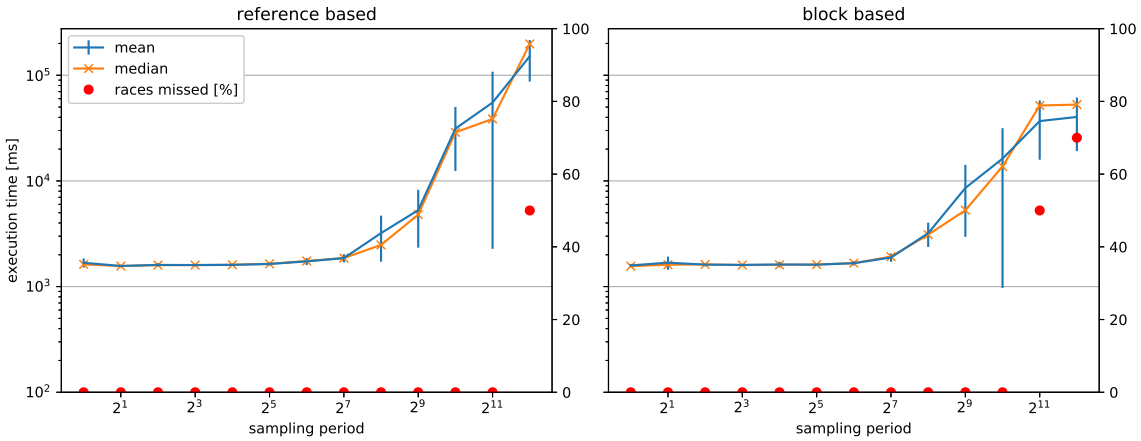


Figure 6.4: Time to find a data race with access probability of $\frac{m}{t} = 10^{-5}$ for various sampling periods. The experiment is executed 10 times each, the error bar denotes one standard deviation.

application with up to 2^{18} rounds. We choose this constraint to limit the total measurement time for our experiments.

For sampling periods larger than $2^{10} \approx 1000$ this is not long enough to find all races. Hence, for practical use cases we recommend to choose a sampling period below this point.

When comparing the reference-based and block-based sampling techniques we see that the overhead of the former is approximately 2 times larger for large sampling periods (where most memory accesses are skipped). This discrepancy comes from the instrumentation that is executed on each skipped memory reference: In the block-based case, a preservation of the flags register is not required. Hence, only a single memory access is required to check the detector state. Additionally, the instrumentation consists of fewer instructions which results in higher L1 and L2 cache hit rates. In contrast, the "reference-based" logic requires preserving the conditional flags and executes more logic. Finally, this results in a higher constant overhead which cannot be avoided by using sampling. This also means that using large sampling rates in combination with block-based sampling just results in worse detection results but not in a reduction of the overhead.

6.3.1 Sampling Techniques

We measured that the block-based approach works best on applications with many function calls, as each call triggers the shadow-stack and hence the state-changing logic. However, in benchmarks and applications with long-running loops without function calls, we missed even frequently occurring races. This is due to the time points when the detector state is changed: Often the only calls made in these long-running loops come from the synchronization mechanisms. These mechanisms are often implemented using multiple call statements,

which distorts the relation between the number of calls and the number of processed memory references. In this case, we observe a large `f-segment`, containing many memory accesses followed by many calls with only few memory references. As the exact distribution of memory references per function call strongly depends on the application, we cannot provide a generic model to calculate the detection probability.

If the interesting code section does not contain any calls (e.g. a loop), we either catch all references or none. This is due to the detector state which is only changed on calls. While we could lessen this issue by also changing the state in the *clean-call* (buffer full) event, this further degrades the results: If we start with an enabled detector, we capture the first n references, then process the buffer and recalculate the sampling decision. This disables the detector and hence, no further references are captured. This implies that no further *clean-call* is executed. Unfortunately, we found no efficient solution to recover from this situation. In future iterations of DRace we plan to investigate this issue further and possibly instrument a small portion of jumps as well.

With reference-based sampling the race detection probabilities are more predictable. However, the overhead of the additional logic that has to be executed on each memory reference is significant. Finally, this increases the minimal overhead we can achieve, even if extensive sampling is used. In our opinion block-based sampling should be used in time critical applications, even if the detection accuracy is less predictable.

Overhead Comparison

Figure 6.5 (left) shows the execution time of the "sampler" mini-app for a growing sampling period. Here, the overhead of the block-based sampling technique is significantly lower compared to the reference-based one for small sampling periods. For larger periods the execution times converge. We explain this through the instrumentation that is added to each memory referencing instruction. Consider the inserted instructions (Listing 6.1) which are executed on both a skipped and a sampled instruction: In the block-based case, skipping an instruction requires just one additional read of the detector state. In contrast to that, reference-based sampling requires one additional read and a write to the control block, which includes the sampling counter.

For references which are sampled, two additional memory accesses are required to update the sampling counter: one to read the next period length and one to store the new control block. This explains why the overhead of this approach grows hyper-exponentially when reducing the sampling period (i.e. processing more events). To ensure that approximately the same number of references is sampled in both settings, we track this number as well. The results are visualized in Figure 6.5 (right). Here we also observe that the variance in the number of sampled references in the block-based setting remains low for sampling periods up to 2^{11}.

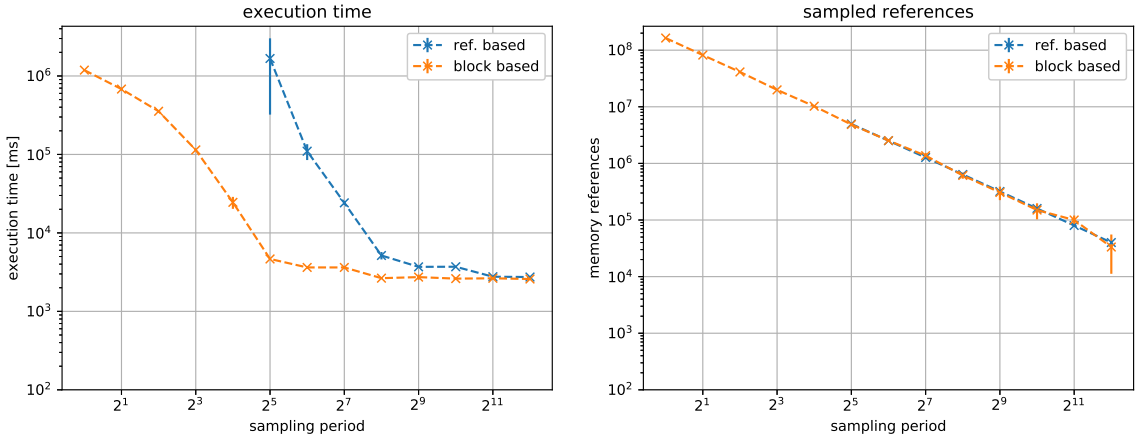


Figure 6.5: Execution time and processed memory references for both reference based and block based sampling.

In real world application we expect the overhead to be smaller, as these applications are not purely memory bound. The evaluated benchmark "sampler" implements the worst case situation, where almost all instructions are reads or writes to the memory.

In future iterations of DRace we plan to further reduce the number of memory accesses by combining multiple values and states in one 64 bit value. Additionally, multimedia or vector registers could be used as most applications do not use them.

6.3.2 Dynamic Scoping

With dynamic scoping we try to exclude high-traffic application parts from the analysis. As these parts have to be located first, each executed fragment of the application is analyzed at least n times before the instrumentation is removed. We further observe that races are mostly detected on the first few executions of a program section. This enables us to analyze every fragment with full instrumentation and to remove the instrumentation later on. We may only remove the memory tracking instrumentation, but not the instrumentation of calls to synchronization logic. As most overhead comes from the tracking and processing of memory references, this is not a limitation. For non-instrumented blocks we measured an execution overhead of $\approx 20\%$ compared to the native execution.

We achieved best results by just using DynamoRIO's internal differentiation between traces and normal fragments. Here, our client does not add any additional overhead as the logic is directly implemented in DynamoRIO. However, we have to admit that this approach is limited to tracking frequent basic blocks, including frequent instruction pointers. What we are more interested in is a combination of instruction pointer and memory location. Otherwise concurrent accesses to dynamically calculated memory locations might not be

processed properly. Unfortunately, we did not find a solution to implement this tracking efficiently.

Nonetheless, we observed that a tracking at the granularity level of fragments is sufficient to find most races. In the case of our mini-apps, this strategy proved to be best as we were able to find all races while achieving the lowest total overhead among all tested strategies. Tests on real world applications are evaluated in the next chapter.

7 Case Studies

In this chapter we evaluate our tool DRace on real-world applications and show that our theoretical considerations also apply in this setting. We further demonstrate the impact of sampling and scoping on the runtime overhead as well as the detection quality. When applicable, we compare our results with the Intel® Inspector XE and show that our tool is superior in both overhead and detection accuracy.

Cluster Similar Races Often, a single data race is detected multiple times. To avoid printing the same race over and over again, the detectors group similar races and just report the first occurrence. This reduces the size of the report and makes it easier for the developer to find the actual root cause. However, when comparing multiple race detectors this clustering becomes a problem as not all detectors follow the same logic. In our case, the only applicable detectors are the Intel® Inspector and the TSan which is included in DRace.

Here, we observed that the TSan treats two races as equal only if both accessed memory entries, rw-mode per access and the first callstack do exactly match. In contrast to that, the Intel® Inspector groups races on the accessed memory and the top-entry of both callstacks only. By that, the number of races which are reported by TSan is significantly higher than what Intel® Inspector reports. For a fair comparison we agree on the Intel® Inspector logic and summarize the TSan reports accordingly.

7.1 Industrial Application

We analyzed an industrial human machine interface application which is written in C++ using the Qt5 Graphical User Interface (GUI) libraries. As the application is confidential, we can neither provide any quantitative numbers nor source code information. Nevertheless we include this application as it shows the extendability of DRace and it serves as an example of how to perform a data race analysis. The general idea is to perform the analysis iteratively. This means, we first observe the found data races, locate and annotate the missing synchronization procedures and re-run the detector. Additionally, we annotate or exclude the benign races.

At first we start with the default instrumentation configuration of DRace and use a long sampling period of 1024 to locate (un-instrumented) synchronization procedures by looking

at the observed races. After every iteration we reduce the sampling-period until we reach a point where the responsiveness of the GUI significantly drops.

Qt5 Synchronization The Qt5 synchronization procedures are implemented in the Qt headers. Hence, the logic is directly bundled with the application using it and executed in the user level. A tracking of the locks using the OS' synchronization API is not sufficient, as the `QMutex` implementation uses a combination of a fast Spinlock and a native lock to improve the performance. Additionally, we do not want to annotate this would require us to either modify the internal Qt sources or to insert code all over the target application.

Instead, we use the debug information of the application binaries to locate the instances of synchronization functions. This is supported in DRace by adding the `aquire` and `release` symbol names to the corresponding entries in the configuration file. Here we obtain the symbol names from the official documentation.[1] We follow this procedure for all other Qt synchronization procedures similarly.

Small String Optimization and Allocators After we added the Qt5 synchronization data races appear that are related to lock-free memory allocators and string classes. These topics are often related, as many string implementations have an optimization for short strings (strings up to an arbitrary length). These small strings are allocated on the stack by using a buffer in the string object. If the buffer is too small, memory is allocated on the heap by possibly using an allocator. In case of custom (user-level) allocators, the allocator has to ensure that no memory is published twice. Often this is done using lock-free data structures. These lead to false-positively detected data races, if their logic is not annotated.

Challenges We found many data races related to efficient string implementations. However, it is tricky to finally decide if a reported data race is actually a correctness issue. In highly optimized implementations often only the team that initially developed the code is able to verify this. Finally, we reported the results back to the development team for further analysis.

This is not a particular problem of DRace, but data race detection in general. Hence, we recommend introducing this technique as early as possible in the development process to annotate these cases. Starting with a race-free code base makes it easier to find newly added races as the user is not tamped with many false positives. Finally, the race detection can be integrated into the continuous integration process.

Performance and Applicability Previous attempts to analyze the target application using the Intel Inspector XE failed, as the application crashed after a few minutes during the startup phase. We suppose that this is related to the memory consumption of the tool

[1] `http://doc.qt.io/qt-5/qmutex.html`

which is higher than the available Random Access Memory (RAM) of the machine. This also applies to the coarsest grained detection and instrumentation settings.

With DRace we were able to analyze this application by using an instrumentation sampling period of 5 and a sampling period of 20. Using this setting, the GUI remained responsive and data races were reported. Additionally, we were able to speed up the analysis by externally disabling the detector during the startup phase. This reduced the startup time from 5 minutes to 30 seconds.

Static scoping turned out to be tricky without deeper knowledge in the architecture of the application, as we were not able to spot memory intense sections. With dynamic scoping we were able to further reduce the runtime overhead by a factor of 2x to 5x at the cost of possibly missing some data races.

7.2 Managed Application

As a proof of concept we analyzed a managed sample Dotnet Core application that demonstrates basic features of ASP.NET Core: The "West Wind Album Viewer" is a showcase application consisting of a backend, written in C# and a web frontend based on AngularJS. The source code is publicly available on GitHub.[2] We choose this application mainly because the communication between the frontend and the backend imposes time constraints.

The application internally uses a database with music albums and provides a user interface to view and modify the data. The database uses "SqLite" and is bundled with the application. Hence, it is instrumented along with the rest of the application. For the performance analysis, we measure the following operations:

1. **Startup**: the time until the web frontend becomes active
2. **Overview**: the album overview page, where all albums are shown including a short description
3. **Album**: a page that shows all information regarding a single album

We run the analysis with the settings shown in Table 7.1. A sampling period of infinity corresponds to a disabled detector

Performance A challenge in this context was the application startup phase, where ≈ 170 modules where loaded. For each Dotnet internal module, debug information is queried using the MSR. This took in average 15 ± 2s in total if the debug information is already cached, which is in average $\approx$ 90ms per module. If the debug information is not in the cache, it is downloaded which takes a few second per module, depending on its size.

[2] `https://github.com/RickStrahl/AlbumViewerVNext`

setting	sampling period	instr. period	mode	exclude stack	scoping
(1)	$[2-64]$	1	fast-mode	yes	no
(2)	∞	1	fast-mode	yes	no
(3)	∞	∞	fast-mode	yes	no
(4)	DynamoRIO without DRace				
(5)	native execution				

Table 7.1: Settings of DRace used in the West Wind Album Viewer evaluation.

We experimented with disabling the detector during this phase which reduced the total startup time from $74s$ to 55s for sampling period $T = 2$. This shows that the major overhead during the startup phase is due to the module handling and instrumentation but not due to the race-detection itself. With $T = 1$ we were not able to run the program, as timeouts of a heartbeat procedure occurred. Figure 7.1 shows the startup time and times for two requests for various settings and sampling rates.

We observe that large parts of the overhead during the starting phase are created by DynamoRIO (setting 4). As DRace uses DynamoRIO, this defines the minimal possible overhead DRace could achieve. The additional slowdown in setting (3) is generated by the analysis and instrumentation of calls (shadow stack, see Section 4.4) and of synchronization procedures (see Section 4.3.4).

Setting (2) defines the baseline for sampling (setting 1), as a sampling period of $T_s = \infty$ means that no reference at all is analyzed. The difference in the overhead between the settings also depends on the workload. During the startup phase, the module processing and instrumentation mainly accounts for the overhead. In the other two scenarios, the overhead is predominantly due to the processing of the instrumented code. Here, we consider only the second execution of each scenario, as the first execution triggers a just-in-time compilation and instrumentation of the newly generated machine code. Additionally, we observe that the effect of sampling is more visible in these scenarios, as now the overhead of the analysis becomes dominant.

Results We found many races that are related to a concurrent queue implemented in `System.Collections.Concurrent`. Here, custom synchronization or tasks are used which is currently not implemented in DRace. The symbol lookup worked correctly and the obtained callstacks where correct. However, inlined functions did not show up as we currently just rely on the shadow stack information (see Section 4.4). This is an issue for large Dotnet applications as they heavily use inlining. In future iterations of DRace, we plan to refine the callstacks by using the `ICorProfiling` API.

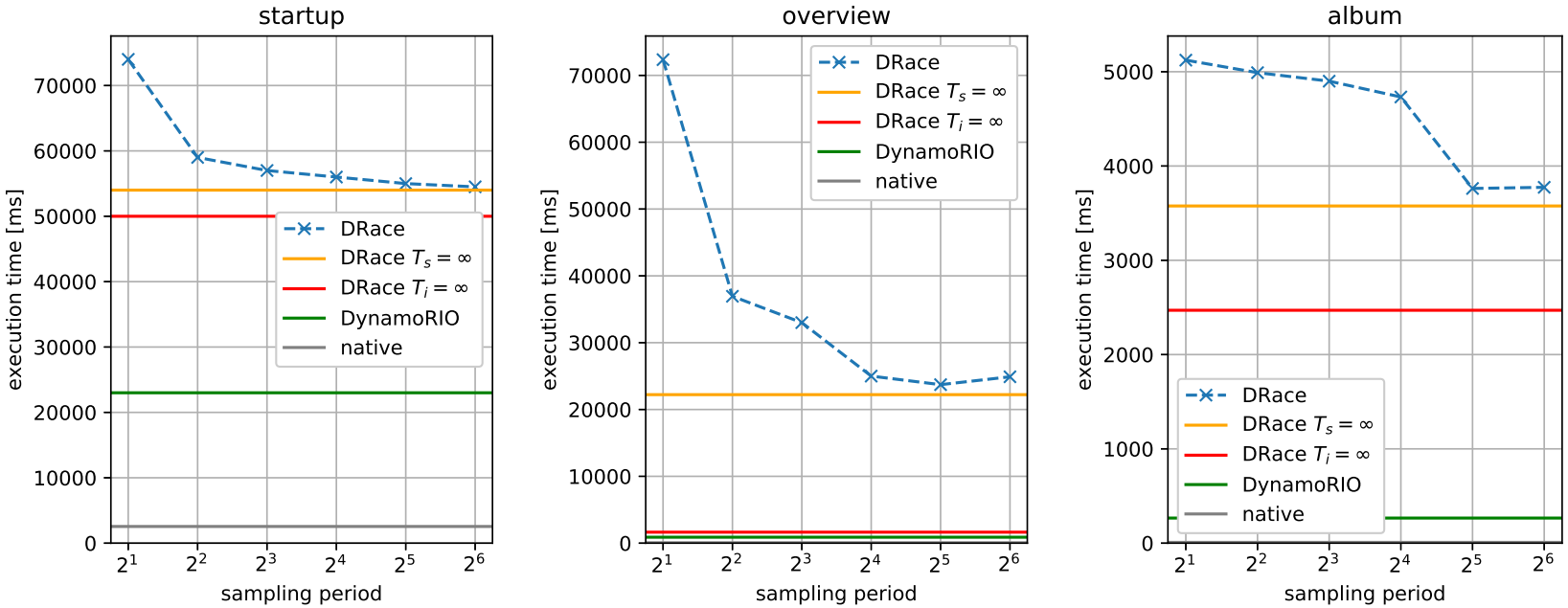


Figure 7.1: Times for the startup and two application requests of the West Wind Album Viewer. For $T_s = \infty$ no reference is sampled, for $T_i = \infty$ no memory access is instrumented.

7.3 Open Source Applications

To show the universal applicability of DRace we run the tool on two open source applications: a publicly available file manager and compression tool "7-zip" as well as a ray tracer ("POV-Ray"). We benchmark this application with with various performance tuning options of DRace to compare the impact on both found races and tool overhead.

The analysis is done on a consumer grade notebook with 16GB of memory and 2 physical CPU cores, using hyper-threading with 2 threads per core. To avoid a colored view on the results, we show the raw number of races our tool outputs. Races are just merged, if they share two identical stack traces. Hence, we might find many races, even if there is actual just one concurrency issue. Additionally, we observed that especially for GUI applications most races are benign races.

7.3.1 7-Zip File Manager

In this test, we use the GUI version ("7zFM.exe") and analyze a predefined click sequence of events. We do that to compare different settings of DRace for the same task, and finally to get reproducible results. Here we compare two common scenarios: extracting a ZIP archive (1) and calculating a checksum of another archive (2). For practical reasons we choose the ZIP archive in a way such that a single measurement takes at most five minutes.

Setting 1: ZIP Extraction

1. start "7zFM.exe"
2. extract `embb1.0.0.zip` (Siemens/EMBB 1.0.0 Release[3])
3. close "7zFM.exe"

Setting 2: Checksum Calculation

1. start "7zFM.exe"
2. open (inspect) `boost_1_68_0.zip` (Boost 1.68.0) Release[4])
3. calculate sha256 checksum of archive
4. close "7zFM.exe"

Preparation

To validate the race detection on this application, we inserted five artificial data races in the adapter between the progress dialog and the extraction algorithms. We did this by removing the critical sections around the accesses. As the protected data is only used to display statistics regarding the deflation progress, the correctness of the extraction process is not affected. For the exact positions in the code, we refer to the Appendix 3.

We evaluate the performance tuning strategies "static sampling" and "dynamic scoping" on this application regarding both runtime overhead and the number of detected races. To

[3] `https://github.com/siemens/embb/releases/tag/v1.0.0`
[4] `https://dl.bintray.com/boostorg/release/1.68.0/source/`

differentiate between the overhead added by the detector and the overhead of the instrumentation itself, we perform all strategies also with the "dummy" detector. Of course, the dummy detector does not detect any races, but it shows that the overhead of the instrumentation itself is small.

Sampling

Figure 7.2 visualizes the effect of sampling on both the runtime overhead and the race detection on the 7-zip benchmark. In setting one, the racy sections are executed ≈ 1200 times per execution. For sampling periods up to $T = 32$, all race-causes are correctly identified. Only the number of different callstacks which result in a race decreases, which is not an issue as the cause is always the same. For larger sampling periods, no data races at all are discovered. In these cases, the data races are missed as possibly unrelated synchronization prevents the Happened Before logic to detect a race (see Sec. 3.1.2, Fig. 3.2). In setting two, the racy sections are less often executed and hence the impact of sampling on race detection is larger. Here we already miss data races if sampling is applied at all. For sampling periods up to $T = 8$ we correctly detect three out of five total data races (Figure 7.2 bottom right).

The overhead during this benchmark ranges from 1.4 to 4x, where 1x means no overhead. Here we discovered that the relatively small overhead is due to the I/O bound task: When extracting the zip-archive most CPU cycles are spent in creating the extracted files on disk. Hence, most of the race detection logic is performed during the otherwise wasted CPU time.

For large sampling periods the overhead converges against a fixed value. We expect that, as the sampling only affects the processing of optional events (see Section 4.1). Both mandatory events and the overhead due to the added instrumentation (see Section 6.1.2) remains constant. When comparing the difference in the overhead between the dummy and the TSan detector, we directly see the cost of processing mandatory events. In the checksum calculation benchmark (setting 2), more synchronization events are processed and hence this difference is larger.

Cross Validation of Sampling Model With the data from the "7-zip" benchmark (setting 1) we cross validate our assumptions made in Section 6.1.1. There, we assumed that for a particular race, 2×10^{-5} of all instructions of a target application are racy. This was already one magnitude more pessimistic than what the authors of [She+11] estimated.

In the "7-Zip" run with sampling period $T = 1$ (no-sampling) we observed 33 races by analyzing ≈ 125 million memory references. With the further assumption that none of these races share a memory location, there must be $33 \times 2 = 66$ unique racy locations. Given that, we calculate $\frac{m}{t} = \frac{33}{125 \times 10^6} \approx 26 \times 10^{-8}$ (for a particular race).

However, we do not know how often a racy location has been processed exactly, but from the difference between the test with $T = 32$ and $s = 64$ (boundary) we estimate the following:

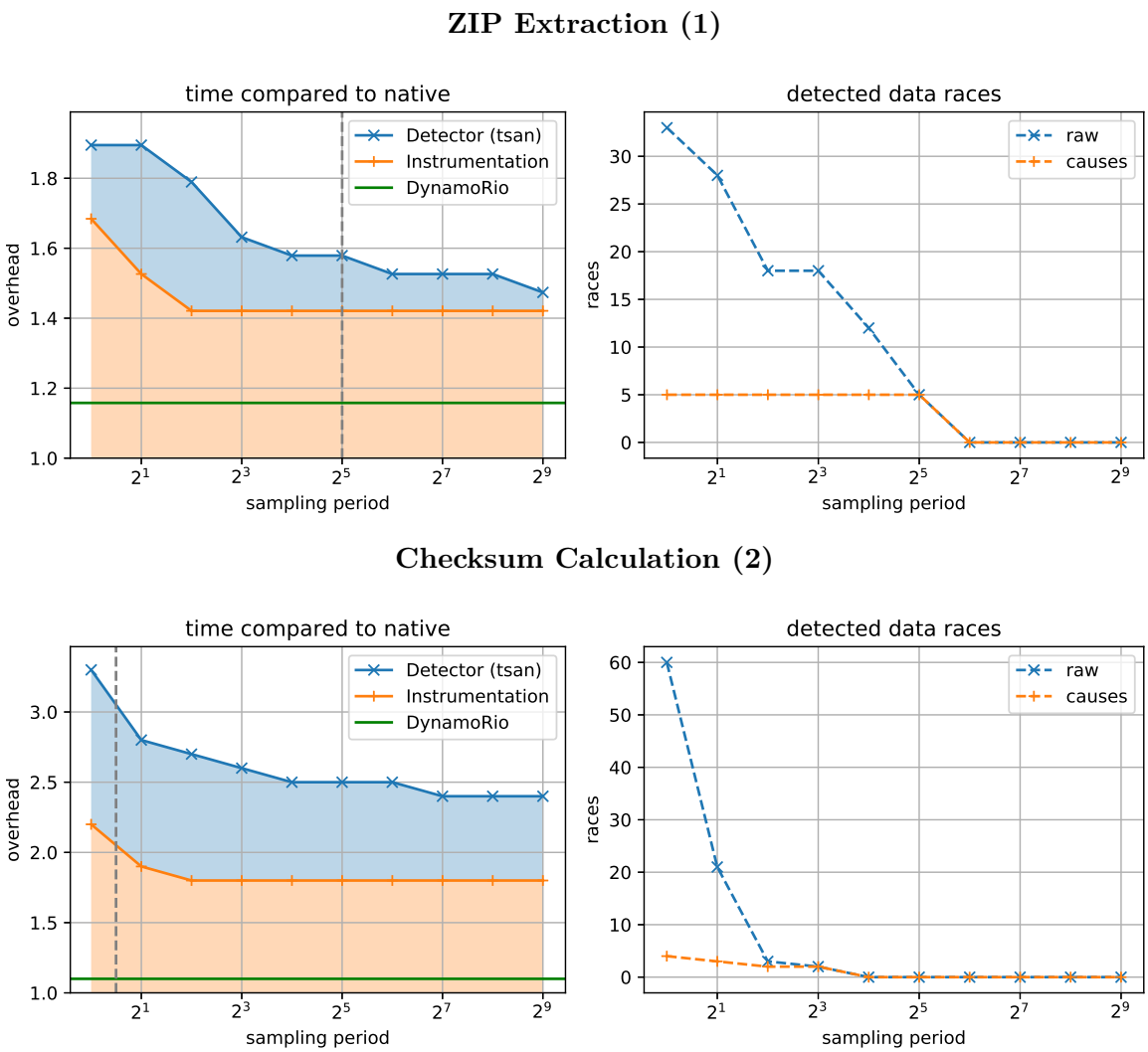


Figure 7.2: 7-zip: relative execution time compared to native execution (left) for various period length. For sampling periods larger than the boundary (dashed vertical line), data race causes are missed. The right graph shows detected races and race-causes.

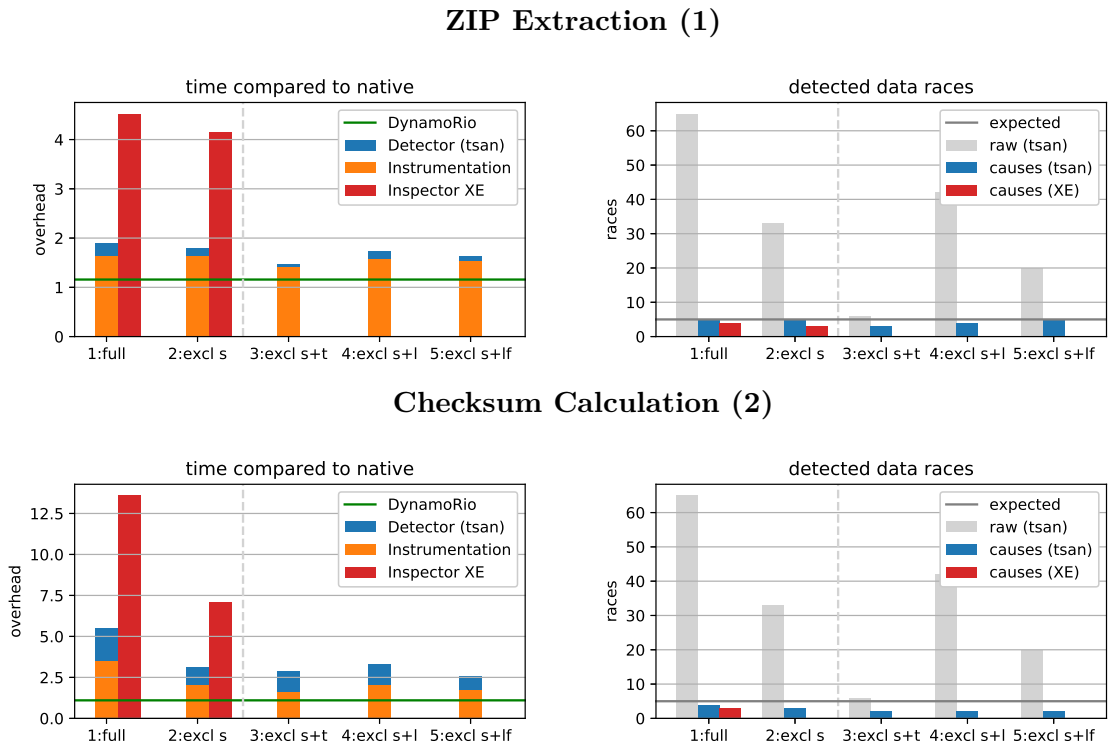


Figure 7.3: 7-zip: relative execution time compared to native execution (left) for various scoping strategies. The right graph shows detected races and race-causes. The first two scenarios are evaluated with Intel® Inspector XE as well.

If we plug the calculated $\frac{m}{t}$ into Equation 6.2, we would expect to catch 40% of all races with $s = 64$. This is obviously not the case (in fact, we did not find any).

Hence, we can conclude that a decent amount of races only occur rarely. A deeper investigation of the benchmark code reveals that 95% of all processed memory references are located in the compression module, which has a much lower race-density. Additionally, the racy GUI updates are triggered by a timer and account only for very few memory accesses per iteration. The synchronization code between two updates of the GUI prevents the Happened Before logic to capture these races if extensive sampling is applied.

This confirms that this model is only applicable to application parts with very long running loops. This holds only partially in this setting, as the GUI update thread runs just for a very short time per iteration. Nonetheless, even in this setting sampling has proved to be a suitable method to reduce the runtime overhead while still maintaining a reasonable detection probability.

Dynamic Scoping

With dynamic scoping we are able to exclude frequent code sections from the analysis after they have been executed a few times. In theory this combines good race detection results

setting	tool	sampl. period	instr. period	exclude stack	scoping
(1)	DRace/ Inspector	1	1	no	no
(2)	DRace/ Inspector	1	1	yes	no
(3)	DRace	1	1	yes	excl-traces
(4)	DRace	1	1	yes	lossy
(5)	DRace	1	1	yes	lossy-flush
(6)	DynamoRIO without DRace				
(7)	native execution				

Table 7.2: Settings of DRace and Intel® Inspector XE used in the "7-zip" scoping evaluation.

with a low overhead. Here we compare the scenarios listed in Table 7.2. The results are visualized in Figure 7.3. For a comparison we also run the benchmarks using the Intel® Inspector XE. Here we use the latest version available at time of this Thesis.[5] For a fair comparison we measure the time beginning at the execution of the analysis until all data race results are reported. Additionally, we use the "use-maximum resources" switch, as otherwise no data races are detected in this setting.

In contrast to the results in Section 6.2.1 we observe that the exclusion of stack accesses does only reduce the overhead in some settings. A inspection of the source code executed in setting one revealed that almost all memory accesses are heap accesses and hence this optimization only affects a small share. The lowest overhead is achieved when traces are excluded (scenario 3). In this scenario code sections which are marked as frequent by DynamoRIO are excluded. While this avoids the overhead of tracking frequent IPs, we have no control on the minimal number of executions of this IP. This also explains why only three out of five data race causes are detected in the ZIP extraction benchmark. In the checksum calculation benchmark, less data races are detected as the executed code uses more synchronization.

The settings (4) and (5) both use the lossy-counting approach (see Section 6.2.1) to identify frequent IPs. This results in more fully instrumented executions before the dynamic scoping kicks in. Hence, we detect more data race causes. The difference between setting (4) and (5) is that the first one does not change the instrumentation but disables the detector during a function starting with a frequent block. In the second case, the instrumentation is removed from all frequent basic-blocks which reduces the memory accesses per execution of this block. Additionally, the scoping is more precise as there is not always a direct mapping between basic blocks and function calls as the basic blocks have a limited size. This explains why setting (4) does only detect four causes while the logically equivalent setting (5) does detect all five causes correctly.

When comparing the overhead between setting four and five, the additional overhead per function call becomes visible: to check if the next `f-segment` is frequent, a lookup in the frequency histogram is necessary. This matches the results in Section 6.2.1 that measured

[5] Version: Intel® Inspector XE 2018, Update 4 (build 574143)

that the `lossy-flush` strategy is more efficient in most cases, especially if the histogram does not change frequently.

Non-Artificial Races

We found two data races which turned out to be benign races. Both races are related to the progress dialog: When an action takes longer than a given time span, a progress dialog is opened. This is done in a second thread, where a Boolean flag is set during the initialization that the dialog has been opened. When the action is finished, it is checked if this flag is set and a message is displayed in the dialog. The accesses to the flag are neither synchronized nor locked. This is ok for x64 systems as loads and stores of a size up to 64 bit are always atomic. In other words, no half-loads happen on these values. However, the visibility of the change depends on the memory model of the CPU as well. For other target architectures like ARM an atomic operation with `memory_order_relaxed` should be used.

7.3.2 POV-Ray Raytracer

The "Persistence of Vision Raytracer" is a 3D software raytracer available on Windows and Linux, known for its photo realistic raytracing capabilities. It is freely available to the public in both binary and source code format.[6]

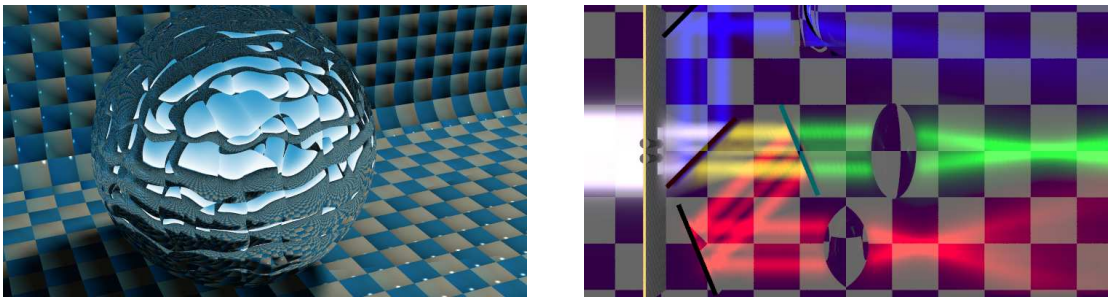
While POV-Ray does not include a 3D modeler, the to-be-rendered scene is specified using a custom scene description language. In contrast to common raytracers, POV-Ray internally represents objects by using their mathematical definitions instead of triangle meshes. This makes it possible to create good-looking scenes with only a few kilobytes in scene source code.

We choose this application as it is known for its highly optimized and compute intensive implementation. This includes a comprehensive usage of the vector and multimedia CPU registers, as well as cache-optimized data accesses. Hence, we expect that possible transparency errors of DRace would show up in this setting. Additionally, we expect the overhead per instrumented memory reference to be higher than normal, as it is unlikely that we find dead registers to avoid the spilling overhead (see Section 4.3.3). Hence, the required registers have to be saved before each inline instrumentation and restored afterwards. This makes it an ideal candidate to both verify the transparency of DRace, as well as to measure the overhead of our tool.

Benchmark Setting

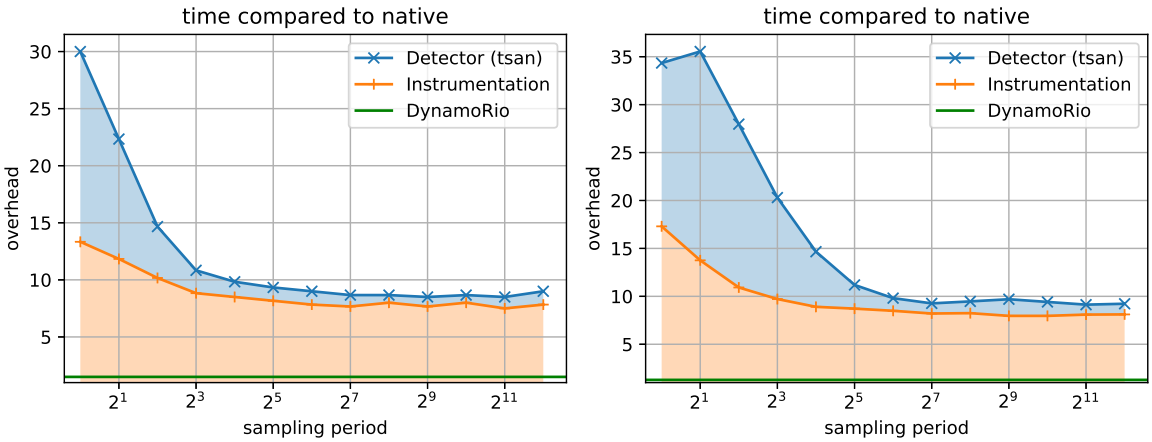
We evaluate the performance of DRace on two scenes that are shipped with the POV-Ray executable. Each scene is rendered with a resolution of 480×270 pixels and an anti-aliasing

[6] `http://www.povray.org/download/`



(a) quilt1.pov (b) optics.pov

Figure 7.4: Scenes that are rendered during the POV-Ray benchmark.



(a) quilt1.pov (b) optics.pov

Figure 7.5: POV-Ray: relative execution time compared to native execution for various period length.

factor of 0.3. Here, we measure the overhead of DRace using both the TSan and the "dummy" detector backend. For practical reasons we exclude all stack accesses from the detection to finally limit the execution time of a single benchmark to 40 minutes. To measure the impact of sampling on the overhead, we evaluate all sampling periods from $T = 1$ to $T = 4096$ which are powers of two.

As we were not able to compile POV-Ray with debug information we do not get symbolized callstacks on the detected races. Hence, we present only measurements regarding the overhead of DRace. According to the raw number of reported data races which resides in the hundreds, we assume that most of them are benign races.

Results

We were able to perform the analysis for sampling periods up to $T = 1$ (no sampling) using both the TSan and the "dummy" detector backend. For a comparison we also tried to run the benchmark using the Intel® Inspector XE, but we did not find a setting where POV-Ray was able to start. With DRace, we did not observe a single application crash during our benchmark measurements.

Figure 7.5 shows the measured overhead of DRace on both scenes compared to native execution. The benchmark execution took 180 seconds for the "quilt1.pov" (a) scene and 1820 seconds respectively for the "optics.pov" (b) scene with sampling period $T = 1$. During the execution, DRace processed $\approx 10 \times 10^9$ (a), $\approx 133 \times 10^9$ (b) memory references.

As the POV-Ray application is compute-bound the analysis of data accesses cannot be performed during idle CPU periods. Hence, the overhead of 8x to 36x (compared to native) is significantly higher than what we observed in the 7-Zip benchmark.

Similar to the 7-Zip benchmark we observe that the overhead converges against a fixed value for large sampling periods. Due to the optimizations in POV-Ray, no dead registers are available to be used by DRace' inline instrumentation. Hence, a constant overhead remains even for large sampling periods as three registers have to be preserved and restored on each memory referencing instruction. Finally, this leads to a minimal overhead of factor 8.

In Figure 7.5 (b) we observe a spike in the overhead at sampling period $T = 2$. To ensure that this is not an outlier, we executed this setting with $T = [1, 2]$ three times with approximately equal results. Finally, this turned out to be due to frequent re-calculations of the sampling decision (see Section 6.1.3) resulting in a higher L2 cache miss rate. Without sampling, less logic has to be executed per call, as the sampling related parts are skipped. This increases the locality of the code resulting in a higher cache hit rate and better performance.

8 Conclusion and Future Work

In this chapter we discuss the previous work and point out future ideas to further improve DRace. Finally, we summarize this thesis.

8.1 Revisiting the Objective

After a short introduction on aspects of data race detection, we specify the goals of this thesis and highlight our contributions (Chapter 1). In Chapter 2, we locate our work in the surrounding field of applications. Subsequently, we outline existing work related to this topic (Chapter 3). Here, we identify shortcomings of existing detectors in the context of large and real-time applications.

In the following Chapter 4, a new data race detector is shaped which provides novel features to approach the shortcomings of the existing tools. Here we discuss the two main goals "Exchangeable Race-Detection Algorithm" and "Customizable Instrumentation": For the former goal, we specify a generic interface to separate the detection algorithm from the instrumentation. Finally, this creates a framework for data race detection which supports multiple algorithmic backends. The second goal focuses on the instrumentation itself: Here we cover strategies to efficiently trace the memory accesses and the related callstacks. Additionally, we present options to extend the instrumentation to work with custom synchronization patterns.

In Chapter 5, managed code applications based on the Dotnet runtime are discussed. We demonstrate that a uniform processing of managed and native code is possible and that cross-code data races are detected properly. Further, we discuss strategies on how to perform symbol resolution on managed code that is running under a dynamic instrumentation framework. Finally, we present an enhanced strategy to handle user-level synchronization in managed code which has not been implemented prior to our work.

Chapter 6 covers a set of sampling and scoping techniques to reduce the runtime overhead of the data race detection. We discuss a theoretical model to predict the influence of sampling on the accuracy of the data race detection as well as multiple strategies to implement the sampling logic. Subsequent, the model and its implementation is evaluated on a sample application. Multiple scoping approaches are discussed to select interesting parts of the target application manually and dynamically at runtime. Finally, we evaluate the strategies on managed and native demo applications regarding overhead and detection accuracy.

The last Chapter 7 covers an evaluation of the previously discussed features of DRace on real world applications. Here, we show that DRace is suited to analyze a native industrial application which is bound by external time constraints. We further evaluate the performance optimization strategies on a managed application as well as on two open source applications and finally show that DRace fulfills the goals specified in the problem statement (Section 1.1).

8.2 Future Work

The next goals are to evolve DRace from the beta state to a tool that can be used productively. One step regarding this is to further optimize the instrumentation:

A good starting point in this context is to optimize the inline instrumentation to reduce the number of additional memory accesses. We further evaluate how to reduce the number of *clean-calls* by combining subsequent calls. This should improve the performance on synchronization heavy applications. Additionally, we work on reducing the complexity and cost of the memory processing logic to further reduce the overhead of the detector. In this context we plan to avoid the pre-processing of events that are pushed into the detector which is currently necessary for the TSan implementation. Last but not least we plan to implement other data race detection algorithms like "FastTrack2" which are optimized for massively parallel applications.

In some corner cases, locks in DRace interfere with application locks which sometimes results in deadlocks. Here we plan to completely remove the necessity for locking by using local and lock-free data structures if possible. Due to the client transparency limitations we cannot use widely available implementations but have to implement custom versions of these data structures.

Currently the support for managed code is in the state of a proof-of-concept. In the future we plan to work intensively on this feature. Together with domain experts we plan to improve this component by adding support for exclusions, improving the symbol lookup and finally reducing the relatively high overhead discovered on managed targets. This is likely to require modifications on the instrumentation framework (DynamoRIO) itself to better support managed code components.

8.3 Summary

Modern applications have become more and more complex which creates a growing need for tools to analyze these. Aspects in this context are correctness issues which are notoriously hard to find with testing. Tools for correctness analysis of multithreaded applications have been used for decades, but mostly limited to native applications and static build processes as well as the unix OS. With modern programming languages that are executed using a JIT,

new tools are required to analyze a target application that is assembled at runtime. This thesis conclusively shows a concept of a data race detector that supports a uniform analysis of managed and native components. We present DRace, a framework for dynamic threading analysis with focus on extensibility and performance tuning. To show the capability to analyze hybrid applications, we extended DRace to support applications with Dotnet based managed components.

One issue in the context of data race detection is the high overhead of the tool which massively slows down the execution of the application. Here we show that a low overhead data race detector is feasible by using sampling and scoping approaches. With DRace, large applications can be analyzed which are beyond the scope of off the shelf applications. External time constraints of the application under test can be met by using a combination of sampling and scoping techniques.

We have made the source code of DRace available to the public, to be used and extended for custom applications. We hope that our tool helps in finding correctness issues in large applications and finally to reduce the number of concurrency related bugs.

Glossary

ABI Application Binary Interface.

API Application Programming Interface.

CIL Common Intermediate Language.

CLI Common Language Infrastructure.

CLR Common Language Runtime.

CPU Central Processing Unit.

CTI CPU instruction that modifies the instruction pointer.

DAC Data Access Component.

DAG Directed Acyclic Graph.

DLL Dynamic Link Library.

fragment sequence of machine-code instructions that terminates with a control transfer operation (e.g. `JMP`, `CALL`, `RET`, etc.).

GCC GNU Compiler Collection.

GUI Graphical User Interface.

HB Happened Before.

IP Instruction Pointer.

JIT Just-in-Time Compiler.

LEA Load Effective Address.

LLVM The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name “LLVM” itself is not an acronym; it is the full name of the project.

LTL Load-Time Locatable code is modified at load time and then must be executed from a particular memory address.

MinGW MinGW, a contraction of ”Minimalist GNU for Windows”, is a minimalist development environment for native Microsoft Windows applications.

MPI Message Passing Interface.

MSIL Microsoft Intermediate Language.

MSR Managed Symbol Resolver.

MT Mersenne-Twister Engine.

NGen The Native Image Generator allows to pre-compile a CLI assembly for a specific target architecture instead of letting the CLR do a just-in-time compilation at runtime.

OpenMP Open Multi-Processing is a API for shared-memory programming using multiple threads in C and C++.

OS Operating System.

PDB Program database (PDB) is a proprietary file format (developed by Microsoft) for storing debugging information about a program.

RAM Random Access Memory.

RNG Random Number Generator.

Spinlock a method to perform mutual exclusion of a code section by repeatedly checking the state of a Boolean variable. The mechanism can be implemented in the user-mode which avoids expensive calls to the OS synchronization procedures. For a C++ implementation see Appendix 1.

TLS Thread Local Storage.

TSan ThreadSanitizer.

Valkyrie Valkyrie is a GUI for the Memcheck and Helgrind tools which enables the user to browse and inspect the analysis reports.

XML The Extensible Markup Language defines an encoding for documents in a format that is both machine and human readable.

List of Figures

Bibliography

[App11] Andrew W. Appel. “Verified software toolchain”. In: *European Symposium on Programming*. 2011, pp. 1–17.

[Atz+16] Simone Atzeni et al. “ARCHER: effectively spotting data races in large OpenMP applications”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 53–62.

[Bey+07] Dirk Beyer et al. “The software model checker b last”. In: *International Journal on Software Tools for Technology Transfer* 9.5-6 (2007), pp. 505–525.

[Blä18] Luc Bläser. “Practical detection of concurrency issues at coding time”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, pp. 221–231.

[Bru04] Derek L. Bruening. “Efficient, Transparent, and Comprehensive Runtime Code Manipulation”. PhD thesis. MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2004.

[Bru18] Derek Bruening. *The DynamoRIO API*. 2018. URL: `http://dynamorio.org/docs/`.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. “Simplify: a theorem prover for program checking”. In: *Journal of the ACM (JACM)* 52.3 (2005), pp. 365–473.

[DS91] Anne Dinning and Edith Schonberg. “Detecting access anomalies in programs with critical sections”. In: *ACM SIGPLAN Notices*. Vol. 26. 1991, pp. 85–96.

[FF09] Cormac Flanagan and Stephen N. Freund. “FastTrack: Efficient and Precise Dynamic Race Detection”. In: (2009).

[FF10] Cormac Flanagan and Stephen 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*. 2010, pp. 1–8.

[FF17] Cormac Flanagan and Stephen N. Freund. *The FASTTRACK2 Race Detector*. 2017. URL: `http://dept.cs.williams.edu/~freund/papers/ft2-techreport.pdf`.

[Hil+13] Tobias Hilbrich et al. “MPI runtime error detection with MUST: advances in deadlock detection”. In: *Scientific Programming* 21.3-4 (2013), pp. 109–121. ISSN: 1058-9244.

[Hol97] Gerard J. Holzmann. “The model checker SPIN”. In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295.

[HP00] Klaus Havelund and Thomas Pressburger. “Model checking java programs using java pathfinder”. In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 366–381.

[Huf54] David A. Huffman. “The synthesis of sequential switching circuits”. In: (1954).

[JT08] Ali Jannesari and Walter F. Tichy. “On-the-fly race detection in multi-threaded programs”. In: *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*. 2008, p. 6.

[KW10] Vineet Kahlon and Chao Wang. “Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs”. In: *International Conference on Computer Aided Verification*. 2010, pp. 434–449.

[KZC12] Baris Kasikci, Cristian Zamfir, and George Candea. “Data races vs. data race bugs: telling the difference with portend”. In: *ACM SIGPLAN Notices* 47.4 (2012), pp. 185–198.

[Lam78] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[Lan07a] Rich Lander. *Data Access Component (DAC) Notes*. 2007. URL: `https://raw.githubusercontent.com/dotnet/coreclr/master/Documentation/botr/dac-notes.md`.

[Lan07b] Rich Lander. *Profiling*. 2007. URL: `https://github.com/dotnet/coreclr/blob/master/Documentation/botr/profiling.md`.

[Luk+05] Chi-Keung Luk et al. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Acm sigplan notices*. Vol. 40. 2005, pp. 190–200.

[Mag+02] Peter S. Magnusson et al. “Simics: A full system simulation platform”. In: *Computer* 35.2 (2002), pp. 50–58.

[Mic17] Microsoft. *Anatomy of a DbgEng Extension DLL*. Ed. by Microsoft Hardware Dev Center. 2017. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/anatomy-of-a-dbgeng-extension-dll`.

[Mic18a] Microsoft. *PE Format*. Ed. by Microsoft Windows Dev Center. 2018. URL: `https://docs.microsoft.com/de-de/windows/desktop/Debug/pe-format`.

[Mic18b] Microsoft. *Synchronization Functions*. Ed. by Microsoft Windows Dev Center. 2018. URL: `https://docs.microsoft.com/en-us/Windows/desktop/sync/synchronization-functions`.

[MMN09] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. “LiteRace: Effective Sampling for Lightweight Data-Race Detection”. In: (2009).

[Muc+99] Philip J. Mucci et al. “PAPI: A portable interface to hardware performance counters”. In: *Proceedings of the department of defense HPCMP users group conference*. Vol. 710. 1999.

[MW07] Arndt Müehlenfeld and Franz Wotawa. “Fault detection in multi-threaded C++ server applications”. In: *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2007, pp. 142–143.

[Nas17] Phil Nash. *Catch2*. 2017. URL: `https://github.com/catchorg/Catch2`.

[NS07] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavy-weight dynamic binary instrumentation”. In: *ACM Sigplan notices*. Vol. 42. 2007, pp. 89–100.

[OC03] Robert O'Callahan and Jong-Deok Choi. "Hybrid Dynamic Data Race Detection". In: (2003).

[Ots+18] Yuto Otsuki et al. "Building stack traces from memory dump of Windows x64". In: *Digital Investigation* 24 (2018), S101–S110. DOI: 10.1016/j.diin.2018.01.013.

[Pat+11] Avadh Patel et al. "MARSS: a full system simulator for multicore x86 CPUs". In: *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*. 2011, pp. 1050–1055.

[Pau94] Lawrence C. Paulson. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media, 1994.

[PS03] Eli Pozniansky and Assaf Schuster. "Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs". In: (2003).

[Sav97] Stefan Savage. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs". In: (1997).

[She+11] Tianwei Sheng et al. "RACEZ: a lightweight and non-invasive race detection tool for production applications". In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 401–410.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer – data race detection in practice". In: (2009).

[SL14] Young Wn Song and Yann-Hang Lee. "Efficient data race detection for C/C++ programs using dynamic granularity". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. 2014, pp. 679–688.

[Sut05] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[TA13] Andrew S. Tanenbaum and Todd Austin. *Structured computer organization*. 6. ed. Boston: Pearson, 2013. ISBN: 0132916525.

[THW10] Jan Treibig, Georg Hager, and Gerhard Wellein. "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments". In: *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. 2010, pp. 207–216.

[Tv07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[Wen17] Maira Wenzel. *Profiling Overview*. Ed. by Microsoft. 2017. URL: https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/profiling-overview.

[Xam08] Xamarin. *Cecil Mono: Bytecode inspector*. 2008. URL: http://www.mono-project.com/Cecil.

Appendix

1 Spinlock Implementation

Listing 1: A spinlock implementation in C++ which does not require a context switch.

```
/**
* Simple mutex implemented as a spinlock
* implements interface of std::mutex
*/
class spinlock {
        std::atomic_flag _flag = ATOMIC_FLAG_INIT;
public:
        inline void lock() noexcept
        {
                while (_flag.test_and_set(std::memory_order_acquire)) { }
        }

        inline bool try_lock() noexcept
        {
                return !(_flag.test_and_set(std::memory_order_acquire));
        }

        inline void unlock() noexcept
        {
                _flag.clear(std::memory_order_release);
        }
};
```

2 Dotnet Concurrent Increment

Listing 2: Demo C# application which concurrently increments a shared counter.

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class param
    {
        public int acc = 0;
    }

    class ThreadCreationProgram
    {
        public static void IncByOne(param p)
        {
            // alternatively use a System.Threading.Mutex
            // or no locking at all for a racy
            // increment
            lock (p)
            {
                ++(p.acc);
            }
```

```
        }

        static void Main(string[] args)
        {
            int numThreads = 2;
            param p        = new param();

           Thread[] threads = new Thread[numThreads];
            for (int i = 0; i < numThreads; i++)
            {
                Thread t = new Thread(() => IncByOne(p));
                threads[i] = t;
            }
            for (int i = 0; i < numThreads; i++)
            {
                threads[i].Start();
            }

            foreach (var t in threads)
                t.Join();

            Console.WriteLine("Value after {0} increments: {1}", numThreads, p.acc);
        }
    }
}
```

3 7-zip Data Race Locations

For the case study in Section 7.3 we use "7-zip" version 18.05 which can be obtained from Sourceforge.[1] We inserted data races into the following functions in file `ProgressDialog2.cpp`:

- `CProgressSync::Set_NumFilesCur`
- `CProgressSync::Set_NumBytesCur`
- `CProgressSync::Set_TitleFileName`
- `CProgressSync::Set_Status2`
- `CProgressDialog::UpdateStatInfo`

[1] `https://sourceforge.net/projects/sevenzip/files/7-Zip/18.05/`