RDS: A Cloud-Based Metaservice for Detecting Data Races in Parallel Programs

Yaying Shi
University of North Carolina at Charlotte
Charlotte, USA
yshi10@uncc.edu

Anjia Wang
University of North Carolina at Charlotte
Charlotte, USA
awang15@uncc.edu

Yonghong Yan
University of North Carolina at Charlotte
Charlotte, USA
yyan7@uncc.edu

Chunhua Liao
Lawrence Livermore National Laboratory
Livermore, USA
liao6@llnl.gov

ABSTRACT

Data races are notorious concurrency bugs which can cause severe problems, including random crashes and corrupted execution results. However, existing data race detection tools are still challenging for users to use. It takes a significant amount of effort for users to install, configure and properly use a tool. A single tool often cannot find all the bugs in a program. Requiring users to use multiple tools is often impracticable and not productive because of the differences in tool interfaces and report formats.

In this paper, we present a cloud-based, service-oriented design and implementation of a race detection service (RDS)[1]to detect data races in parallel programs. RDS integrates multiple data race detection tools into a single cloud-based service via a REST API. It defines a standard JSON format to represent data race detection results, facilitating producing user-friendly reports, aggregating output of multiple tools, as well as being easily processed by other tools. RDS also defines a set of policies for aggregating outputs from multiple tools. RDS significantly simplifies the workflow of using data race detection tools and improves the report quality and productivity of performing race detection for parallel programs. Our evaluation shows that RDS can deliver more accurate results with much less effort from users, when compared with the traditional way of using any individual tools. Using four selected tools and DataRaceBench, RDS improves the Adjusted F-1 scores by 8.8% and 12.6% over the best and the average scores, respectively. For the NAS Parallel Benchmark, RDS improves 35% of the adjusted accuracy compared to the average of the tools.

Our work studies a new approach of composing software tools for parallel computing via a service-oriented architecture. The same approach and framework can be used to create metaservice for compilers, performance tools, auto-tuning tools, and so on.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Software and its engineering** → **Software notations and tools**.

KEYWORDS

Data Race Detection, Parallel Computing, Metaservices, Microservices, Cloud Service API

[1]Source code https://github.com/RaceDetectionService/RaceDetectionService_Server

1 INTRODUCTION

In parallel programming, a data race happens when two or more threads access the same memory location that is not protected by synchronization mechanisms such as memory atomic, compare-and-swap and lock, and at least one of the accesses is a write access. Data race bugs are notoriously hard to find due to non-deterministic behaviors of parallel executions. They may cause severe damage to a parallel application such as crash of program execution, and corrupted results even if the program appears to complete normally.

To find and remove data race bugs in parallel programs, users often resort to data race detection tools. Some of these tools use static program analysis techniques [13, 28]. However, the effectiveness of static analysis tools is limited due to unknown program semantics at compile-time, especially for code using pointers. It is also difficult for the tools to enumerate all possible execution scenarios by static analysis. Thus the rate of reporting false positives and false negatives could be very high [16]. Many other tools [8, 9, 12, 18] rely on dynamic race detection techniques by collecting memory access information of multiple threads at runtime and then analyzing the memory access to detect the data race. Because these tools analyze actual memory accesses of program execution, they can reduce false positives and achieve high accuracy of detecting races. There are also hybrid tools [18, 27, 34] that try to combine the best traits of both static and dynamic analysis techniques.

In reality, data race detection tools vary significantly in terms of the prerequisite software (e.g. compilers and runtime systems), accuracy of detection, supported parallel programming interface, robustness, user interface and the format of reports. A dedicated benchmark suite, DataRaceBench [21], has been developed to compare data race detection tools that support analyzing OpenMP program execution. A report [25] using the DataRaceBench to evaluate four selected tools has been produced to list the strengths and limitations of each tool. For example, ThreadSanitizer [18] and Archer [9] have more efficient time and memory usage. Intel Inspector [8], which is a commercial tool, generates more friendly reports than those by other tools and it also provides a GUI interface. Research tools (Archer and ROMP [12]) reduce some false positives, but they usually have more compile and runtime errors. The finding is that no single tool can deliver the best result for a wide range of data race scenarios across a large collection of multi-threaded programs. Ideally, users would like to have combined results from multiple tools, leveraging the strengths of each

tool. Yet most of those tools require specific compiler tool chains, and it takes a significant amount of effort to install, configure and properly use a single tool. Different tools also have different user interfaces and use different formats to present their results. There is a lack of efforts for defining common APIs for the interface and report format, and about how to efficiently and correctly merge conflicting data race detection results.

In summary, users may encounter three challenges when utilizing data race detection tools. First, it is time-consuming for users to set up software environments for running individual data race tools. Second, it remains unclear how to maximize the accuracy when combining multiple race detection tools generating different or even conflicting results. Lastly, due to the growing complexity of tools, it is urgent to develop a convenient architecture for users to access and utilize each individual tool. For instance, users can use a simple and unified command line to invoke all tools. Meanwhile, they can access the tool server on most electronic devices remotely.

In this paper, we present RaceDetectionService (or RDS), a cloud-based metaservice aimed at providing convenient and high quality data race detection results for parallel programs. Our work has the following contributions:

- RDS integrates four dynamic race detection tools and provides a convenient cloud-based service for users to detect data races of parallel programs. Users only need to upload their code using the provided REST API. The cloud service automatically invokes each tool, aggregates their outputs, and generates a combined report for users.
- RDS provides a generic, two-level composable service framework that can be used for combining multiple tools. A low-level microservice layer encapsulates the functionalities of individual tool, and a high-level metaservice layer aggregates results from the microservice layer. Using the container technology, the framework is extensible to include more tools and to add more layers if necessary. The approach and framework can be used to create cloud metaservice for a wide range of commodity and commercial tools for parallel computing such as compilers, performance analysis tools, auto-tuning tools, etc.
- A unified set of language-neutral REST API functions and input/output data formats in JSON have been developed for the communication between different layers of RDS. Those APIs and data formats facilitate composing and comparing tools of similar functionalities, without the need to customize tool chains and scripts for each individual tools.
- We propose and compare a set of policies that the metaservice uses to merge potentially conflicting results from multiple data race detection tools. The policies include standard set operations, voting policies and OpenMP-specific weighted policies.

We evaluate the quality and overhead of RaceDetectionService by comparing it with selected individual tools using two benchmark suites. Overall, RDS delivers more accurate results with much less effort from users, when compared with the traditional way of using any individual tools. Using four selected tools and DataRaceBench, RDS improves the best and average Adjusted F-1 scores by 8.8% and 12.6%, respectively. For Nas Parallel Benchmark, RDS matches the best accuracy of the four tools. It improves by 35% of the average adjusted accuracy compared to the tools. Using RDS introduces time overhead ranging from 0.5% to 7.05% compared with using individual tools, caused mostly by the post-processing of tool-specific output at the microservice layer. The space overhead of RDS is 1.84% and acceptable.

The remainder of the paper is organized as follows: In Section 2, we present some background information that motivates our work. Section 3 includes details of our design, approach, the implementation of the framework and RDS. In Section 4, evaluation of the RaceDetectionService using benchmarks with regards to individual tools is presented. Finally, we discuss related work in Section 5 and conclude our work in Section 6.

2 LIMITATION AND CHALLENGES OF USING INDIVIDUAL TOOLS

Data races are notorious in parallel computing since they may either crash an entire program or silently corrupt data and lead to unnoticed wrong results. Due to the non-deterministic nature of multithreaded program execution, it is also challenging to detect or debug data race bugs. Numerous tools have been developed to detect data race bugs. They largely fall into two categories: static analysis tools and dynamic analysis tools.

Our work focuses on leveraging existing dynamic data race detection tools to build a race detection service. A dynamic tool works as follows. It first instruments and executes a target program to collect execution traces that include load store information about memory locations accessed by threads. After the execution, traces are analyzed for finding memory accesses that are to the same memory location but from different threads. Techniques such as lockset analysis [30] and identifying happens-before relations [20] are commonly used. The accuracy of dynamic tools depend on many factors such as input data sets, the compilers, the number of threads, the runtime thread schedulers, and so on. These tools may generate false negatives if the instrumented executions do not activate the right execution conditions that lead to data races.

There are four tools that are relatively popular, Intel Inspector [8], Archer [9], ThreadSanitizer [18], and ROMP [12]. However, the current practice of using these tools has not been productive due to the differences and limitations of these tools. It requires significant manual efforts and expert knowledge for users to analyze a given parallel application. This practice has the following major limitations:

First of all, properly installing and configuring a tool is not easy. Most existing detection tools depend on specific compilers and runtime libraries to work properly. These dependencies can be specific in software version numbers and internal flag settings. For example, Archer depends on the LLVM compiler and an OpenMP runtime with OMPT support [29]. However, the OMPT support may not be turned on by the default installation of the OpenMP runtime. It is inefficient for users to build and install new compilers and runtime libraries for a single data race detection tool.

Second, tools have very different user interfaces and command line options, in either GUI or command line interface or both. For instance, Intel Inspector provides command options to control its analysis scope, memory granularity, stack frame depth, analysis

resource levels, and so on. The default setting of these options often leads to suboptimal results according to the previous studies [21]. It becomes a burden for users to understand different options and the consequences of turning on or off certain options.

Third, each tool has its own limitations of detecting races of different types and behaviors as reported by previous studies [12, 21, 24, 25]. Recent research tools (e.g. Archer and ROMP) can reduce some false positives, but they usually have more compile and runtime errors. Commercial tools such as Intel Inspector are often robust with more user-friendly output, however it may not have cutting-edge techniques leading to the best accuracy. In terms of supporting OpenMP programs, these tools vary significantly in terms of the version and constructs of OpenMP. ThreadSanitizer and Archer have lower overhead in time and memory consumption than Intel Inspector[24]. Intel Inspector has user-friendly reports and consolidates multiple data race warnings caused by a same source location[21]. The finding is that no single tool can deliver the best result for a wide range of data race scenarios across a large collection of parallel programs.

Observing the limitations of the individual tools, we are motivated to create a cloud-based framework that combines the strength of multiple tools.

3 DESIGN AND IMPLEMENTATION OF RACEDETECTIONSERVICE

We thus created the RaceDetectionService that employs a service-oriented architecture for its design for the integration of individual tools. The framework follows best practices for cloud service development including service containerization, standard cloud APIs and data exchange formats, and cloud deployment, making the framework easy-to-use, composable, extensible and robust. The resulting framework can be used to create other service-based tools such as compiler and performance tools.

3.1 Service-Oriented Architecture Design

Service-Oriented Architecture (SOA) is a well-established software design pattern for creating services in distributed computing environments. Services communicate via standard protocols, such as HTTP. This design has many advantages including providing open service interfaces, facilitating service composition, ease for remote accesses, abstracting the complexity of individual components, tolerating individual component's failures, and so on. SOA software often leverages cloud, serverless computing, and containers for deployment.

The overall architecture for RDS has a two-layer design that consists of a metaservice layer and a microservice layer. Figure 1 and 2 show the two-layer design, respectively. Both layers use a uniform set of APIs and data exchange formats to communicate requests and results between clients and services. RDS' web interface accepts user inputs related to source code, tool selection, aggregation policies and so on. Additionally through the defined service API, users can also send command line requests to the server and obtain data race detection results, without worrying about how to install individual tools or how to properly combine their results. The system uses multiple loosely coupled docker containers and can be deployed on a cloud server, a private server, a local machine or a personal laptop. Each container is used to implement one service, either the metaservice combining results or a microservice encapsulating an individual tool.

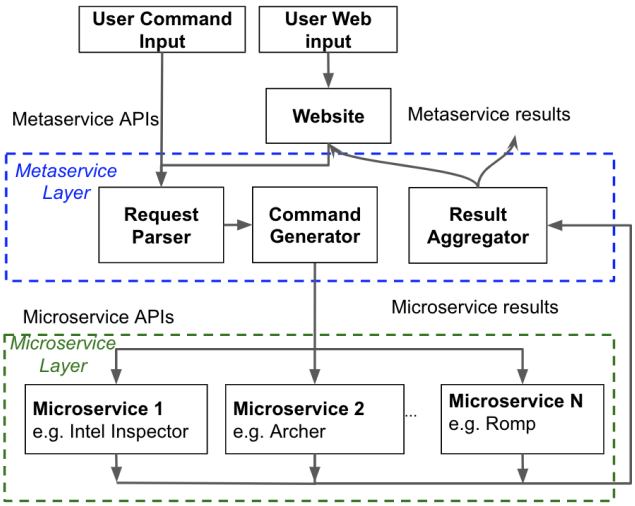

Figure 1: Design of RaceDetectionService

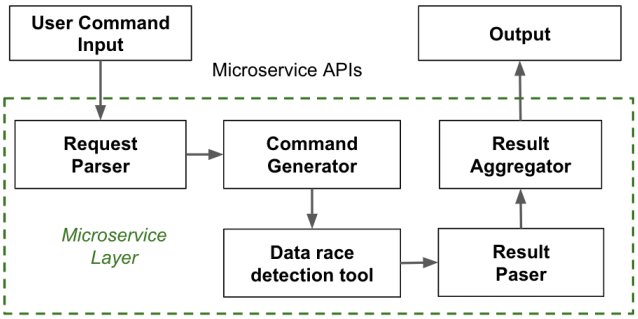

Figure 2: Design of a microservice

3.2 Functionality of Metaservice and Microservice

The metaservice layer, shown in Figure 1 is the service frontend from which users submit race detection requests. The metaservice processes a user's request, generates new service requests with a set of commands for each microservice tool, then sends the service requests to the corresponding microservices. The microservices that encapsulate individual race detection tools process the requests from the metaservice, and respond to the metaservice with the service results. The results are then aggregated using defined policies and presented to users.

The microservice layer is very similar to the metaservice layer except it processes detection requests to a single tool. The APIs and exchange data format are the same as the metaservice layer, thus it allows the user to initiate a race detection request directly to a microservice without going through the metaservice layer. The motivation of this design is to allow flexible deployment so users can use the same API to request metaservice or a specific microservice.

Internally, detection service requests are processed to generate command lines that are specific for the tool the microservice serves. The microservice layer is also responsible for parsing the output generated by a tool and for producing the result in the standard data exchange format. Note that a result aggregator is also needed at the microservice layer, in order to validate and combine results from multiple runs of each tool. By using this loosely coupled service-oriented architecture for the RDS framework, an individual microservice can register itself to the metaservice and contribute to the final output for users. With the benefits of low coupling and high cohesion, an individual microservice can be easily attached or removed from the metaservice layer.

3.3 APIs for Metaservice and Microservice

Table 1 shows the REST API for RaceDetectionService. The API supports three key services: race detection services, user requests, and reports. The API for */reports* are very similar to those for */requests* and are omitted in the table.

The designed API uniformly encapsulates the functionalities of both metaservice and microservice layers that are exposed to the service clients, who might be human or another software component. The uniform design of API simplifies the system and future integration. Human users usually use the API to send requests to the metaservice. They can also opt to directly interact with the individual microservices for debugging or other purposes. The metaservice layer uses the same APIs to invoke microservices in order to obtain results.

Both the metaservice and individual microservice are identified by unique service IDs. For example, the metaservice's ID is "meta" and the microservice for a tool has an ID named "micro-ToolName". Users can use the *GET* method to query available services and obtain their service IDs. A *POST* method to a specified service ID is to send a race detection service request. Users need to provide a file and options when sending the service request. We support both synchronous and asynchronous use of the services, indicated by the *SyncFlag* option. For synchronous use of a service, users wait for the service to complete the actual race detection work and return a report. In contrast, for asynchronous use, users obtain a request ID first while the service does the actual work in the background. The users query status and obtain a report with the request ID later.

Extra options for users to customize desired race detection requests are provided, and listed below:

- *Repeat=int-value* is used to indicate how many times each individual tool should run. If not specified, the default value is 3. This option is intended for microservices, and can be directly passed to microservices or indirectly through the metaservice.
- *AggregatePolicy=policy-name* is used to instruct the aggregator which policy should be used for combining results from multiple microservices or from multiple runs of a microservice tool.
- *SelectMicroservices=s1,s2,...* is useful to customize the internal microservices being used for the metaservice. By default, all available microservices are used.
- *ToolOption="toolName:optionString"* is used to pass tool-specific options to microservice tools. ToolName can be one of the actual tools we integrate into our system, such as IntelInspector, ThreadSanitizer, Archer, and ROMP.
- *BuildOption="optionString"* tells RDS how to build an uploaded file package (zip or tar.gz format). The option string contains the type of the build system (automake or cmake), build target (binary executable), and other build and execution options.

RDS accepts single source files or multiple source files packaged into a zip or tar.gz file. The supported build systems include makefile and cmake. Users should use standard compiler, link and flag names in makefile (such as CC, CLINK, CFLAG, etc.) so RDS can automatically replace them with tool-specific values. Additional build options can be provided to specify the target of build, command options, and so on.

As an example, the following request asks the metaservice to detect data races for an input file package: `curl -X POST -F "file=@NPB.tar.gz" "/RDS/meta?SyncFlag=false&AggregatePolicy=Union&BuildOption=makefile%3ALU%20CLASS%3DS"`. This request specifies that the input file package should be built using a makefile with a make target LU and a build option of a small class. The union policy is used to aggregate results.

3.4 JSON-based Data Exchange Format

JSON is a commonly used data format for cloud service and REST API. Figure 3 shows an example report of data race detection, in a two-column JSON format. The defined JSON data format is used to represent data race detection results for a given program and is used for both metaservice and microservices. The format has the following key components:

- The file starts with a key-value pair to encode the program executable information.
- Data races found in this program are stored as an array object, which in turn is the value of the key named data_races. The array object can be empty if there are no data races found.
- Each array element encodes a pair of memory accesses, followed by the information about individual microservice's results for this pair.
- Each memory access contains its access type (read or write), source code location in the form of file_name:line_number:column_number), and a variable name (symbol).
- The policy used by the result aggregator is also provided as a key-value pair.
- The original raw tool output is also available at downloadable URLs in JSON as well.

3.5 Result Aggregators

The result aggregators are designed to combine results from multiple microservices (used in Figure 1), or multiple runs by a microservice tool for the same detection request (used in Figure 2). The reason for allowing a microservice tool to execute multiple times for the same detection request is because often a dynamic analysis tool needs to inspect each input program multiple times (3 to 5), each with the same or different number of threads to detect data races. Therefore, the results from multiple runs also need to be aggregated. Both aggregators produce the results in the defined

HTTP Method	URL	Parameters	Description
GET	/RDS	N/A	List available race detection services' IDs, such as meta, micro-archer, micro-romp, etc.
POST	/RDS/service-id	SyncFlag, file, options	Send a file to a race detection service specified by its id, with extra options. The service will finish all the work and return a report if the synchronous flag is set to true. Otherwise, the service will immediately return a request ID and a key to authenticate possible HTTP DELETE requests. The actual race detection work will be executed in the background.
GET	/requests	N/A	List all requests submitted to all services
GET	/requests/request-id	N/A	Check the status of a specific request, return a status of nonexist, finished, pending, running.
DELETE	/requests/request-id	key	Cancel an ongoing request, return a status of nonexist, success or failure.

Table 1: RaceDetectionService's API, accessible by curl, Java, Python and other library or tools handling HTTP requests.

```
{
  "program": "a.out",
  "data_races": [
    {
      "read": {
        "location": ["file1.c",64,12],
        "symbol": "A[i]"
      },
      "write": {
        "location": ["file1.c",64,11],
        "symbol": "A[i]"
      },
      "microservices": [
        {"Archer": true},
        {"ROMP" : true},
        {"ThreadSanitizer": true},
        {"Inspector": false}
       ]
    },
    {
      "read": {
        "location": ["file2.c",132,7],
        "symbol": "b"
      },
      "write": {
        "location": ["file2.c",246,31],
        "symbol": "b"
      },
      "microservices": [
        {"Archer": true},
        {"ROMP" : false},
        {"ThreadSanitizer": true},
        {"Inspector": true}
      ],
    }
  ],
  "raw_output": [
       {"Archer": "shorturl.at/uzJR7"},
       {"ROMP" : "shorturl.at/enwH4"},
       {"ThreadSanitizer": "shorturl.at/otO67"},
       {"Inspector": "shorturl.at/dH389"}
  ],
  "aggregate_policy": "Union"
}
```

Figure 3: An example JSON report for discovered data races

JSON format mentioned in the previous section. An aggregation policy is used to combine the results.

Policy	Description
Union	Union of the results for the same data race instance from multiple input reports, which means there is a data race if at least one of the reports says so.
Intersection	Intersection of all results, which means there is a data race only if all reports say so.
Random	Randomly pick a result from an input report
Majority Vote	Simple majority vote. This policy has two variants: 1) the result is positive if there is a tie, or 2) the result is negative if there is a tie.
Weighted Vote	Different tools have different weights based on F1-score measured by DataRaceBench. This policy also has two variants: tie to positive or tie to negative.
Directive-Specific Weighted Vote	Tools perform differently for different types of OpenMP directives. We divide DataRaceBench into different types of directives and calculate each type's F1-scores for each tool.

Table 2: Aggregation policies by the metaservice aggregator

Table 2 shows a list of aggregation policies used by the metaservice's aggregator to combine results from multiple microservices, including set-union, set-intersection, majority-voting, random-selection, weighted-selection, and OpenMP-construct specific voting. The goal is to use a policy which can deliver the best report. If every microservice provider is equally trustable for their results, the naive majority vote should work well. However, the majority vote will not pick the right result if the majority of the selected tools generate wrong results. Another policy is to assign different trustworthy weights to different tools, then we can use weight-adjusted majority vote. Some tools may perform better than the others for a given type OpenMP constructs. There is a policy to give more weight to a tool's results if an input code has a certain OpenMP constructs. If in the future a much larger data set is available, some

learning or heuristics-based techniques might be applied for the final evaluation.

Within a microservice, if a tool can detect a data race at least once, we consider that it has successfully detected the data race. Therefore, we use the union policy for the result aggregator within a microservice.

3.6 Security and Scalability

To ensure the security of our server, we use the rootless docker-in-docker (DIND) technique [5] to run the tools. Rootless DIND is the latest feature of docker since version 19.03. Without exposing root privilege, the docker container is more secure [11]. The code from users is compiled and run by a non-root account in the sandbox. Therefore, harmful activities are limited within the sandbox.

When RDS is deployed on commercial platform providers such as AWS, it takes advantage of the auto scaling feature of the cloud platform [2]. For example, AWS can monitor and maintain the utilization rate of CPU and memory. If AWS notices that RDS has been using 100% of CPU for a period of time, it will increase the allocated CPU resources of the RDS server automatically to lower the rate to a predefined value, such as 75%. Also, we make the entire open-source framework public[2] so it can be deployed by users, which avoids a single centralized deployment with huge operation costs.

4 EVALUATION

We used two benchmark suites, DataRaceBench[21] and NAS Parallel Benchmark (NPB) [14], to evaluate RDS. DataRaceBench is a C/C++ benchmark suite designed to systematically and quantitatively evaluate the effectiveness of data race detection tools for OpenMP programs. NPB is a popular scientific computing benchmark suite to evaluate the performance of parallel supercomputers[14]. The results of RDS are compared to the results generated by using four selected individual tools. We also evaluate the overhead of RDS in terms of both execution time and storage usage.

4.1 Metrics for Evaluation

To evaluate the quality of data race tools, we use five standard metrics: Recall, Specificity, Precision, Accuracy, and F1 score. These metrics are calculated based on four possible results of a tool: True Positive (TP) , False Positive (FP), True Negative (TN), and False Negative (FN) as shown in Table 3.

We also introduce a metric called Test Support Ratio (TSR) to measure the percentage of input programs a tool supports. TSR is used to assess the robustness of a tool. A tool is said to support a test program if the compiler used with the tool can correctly compile the code, and the generated executable can be analyzed by the tool without errors. Otherwise, the tool has one count of unsupported tests. We further provide four categories if a tool does not support a test: compile time unsupported language feature (CUN), compile time segmentation fault (CSF), runtime timeout (RTO), and runtime segmentation fault (RSF). Based on the TSR, we also define a metric called Adjusted F1 score, which equals to an F1 score multiplied by TSR.

[2]https://github.com/RaceDetectionService/RaceDetectionService_Server

4.2 Selected Data Race Detection Tools

We implemented RDS microservices on the top of four available data race tools: Intel Inspector [8], Archer [9], ThreadSanitizer [18] and ROMP [12].

The versions of the tools and compilers used are listed in table 4. Whenever possible, we use the most aggressive settings of a tool in order to generate best possible results, with a potential cost of high overhead. For ThreadSanitizer, we used the clang (v10.0) with compilation flags '-fopenmp -fsanitize=thread -fPIE -pie -g'. As reported by Lin et al. [25], ThreadSanitizer can effectively support OpenMP codes when using the LLVM OpenMP runtime configured with the `LIBOMP_TSAN_SUPPORT` turned on. For Intel inspector, we used the 2010 version with the flag '-collect ti3 -knob scope=extreme -knob stack-depth=16 -knob use-maximum-resources=true'. Archer requires Clang/LLVM 6.0 to compile a program for detection. For ROMP, we used GCC (v9.6.0) to compile the file with flag '-g -fopenmp -lomp'.

4.3 DataRaceBench v.1.3.0

DataRaceBench is a benchmark suite designed to systematically and quantitatively evaluate the effectiveness of data race detection tools. It includes a set of microbenchmarks with or without data races. We used DataRacebench v.1.3.0 [22, 33] for our evaluation. It includes 172 C and 166 Fortran microbenchmarks. The evaluation experiment has the following configurations: the OpenMP thread count is set to be 2, 4, and 8. The size of the array was set to 32. While more threads and larger arrays can be used, it is our intention to use only necessary threads and input data for the tools to effectively discover data races, in order to save cloud resources. We have found using more than 8 threads or larger arrays does not generate noticeable better results. DRB006 is an exception because it is specially designed to trigger data races only when 32 threads are used. Each program of the benchmark suite runs 5 times. If a data race was reported in one of the 5 iterations, we consider the result as positive for the given tool. We run experiments on a single node service with a single Intel(R) Xeon(R) CPU with 16 cores (32 threads) on it.

For RDS, we used different aggregation policies (shown in Table 2) to generate the final results. Since there is an even number (four) of microservice tools, there could be an equal vote situation for Majority vote policy. In this case, RDS uses two different tie breakers. For the weighted and directive specific weighted vote policies, RDS use floating-point weights, which are much less likely to have the equal values so we do not need to handle ties in our experiments.

The directive-specific weighted policy uses the F1 score of each tool for different types of OpenMP directives. We collected such F1 scores in Table 5 based on DataRaceBench's results generated by using individual tools. It is interesting to observe that most tools work relatively well with classic parallel directives, but less so for newer SIMD or target directives. Three of the tools have limited support for the threadprivate directive. Some OpenMP directives only have a few test programs so calculating their F1 score is infeasible. In that case, we use the tool's accuracy as an alternative. For the atomic directive, there is only one test file. Tsan generates false positive for that file and we can not calculate the F1 score for

Tool Result	Ground Truth		Recall	Specificity	Precision	Accuracy	F1 Score
	True	False					
True	TP	FP	TP / (TP + FN)	TN / (TN + FP)	TP / (TP + FP)	(TP+TN) / (TP + FP + TN+ FN)	2 * (P * R) / (P + R)
False	FN	TN					

Table 3: Definition of metrics (Recall, Specificity, Precision, Accuracy and F1 Score)

Tool	version	Compiler
Inter Inspector	2020(build 603904)	Intel Compiler 19.1.0.166
ThreadSanitizer	10.0	Clang/LLVM 10.0
Archer	release_60	Clang/LLVM 6.0
ROMP	version 2	GCC 9.6.2

Table 4: Data Race Detection Tools: version and compiler

it. Thus its accuracy value 0 is used to represent the tool's weight for this directive.

Directives	Archer	ThreadSanitizer	Intel Inspector	ROMP
Parallel	1.00	0.93	0.92	1.00
Parallel for	0.92	0.77	0.72	0.98
Parallel section	1.00	0.67	0.67	0.67
task	0.88	0.36	0.78	0.78
task loop	0.67	0.67	1.00	1.00
simd	0.50	0.50	0.50	0.50
threadprivate	1.00	0.57	0.67	0.60
master	1.00	1.00	1.00	1.00
target	0.50	0.80	0.33	0.80
flush	1.00	1.00	1.00	1.00
single	1.00	1.00	1.00	1.00
atomic	1.00	0.00	1.00	1.00

Table 5: Directive-specific F1-score of the Tools

We have observed that the latest tools have been significantly improved compared to previous studies which reported many compile-time or runtime errors in those tools [23, 25]. ThreadSanitizer hanged when 8 threads were used for some test cases (such as DRB043 and DRB044). But it is considered as supporting all tests since we applied union policy for multiple runs.

The overall evaluation results are shown in Table 6. We use bold font to highlight the numbers that represent the highest quality for each metric in two sets: 1) individual tools and 2) RDS variants using different aggregation policies. Note that for FP and FN, the lower are the better, and for other metrics, the higher values are better.

Out of all the individual tools, Intel Inspector has the best performance based on the adjusted F-1 score. If we consider supported tests only, ThreadSanitizer has the best results for specificity, accuracy, precision, FP and TN. Archer has the best performance for FP, specificity and precision. Romp has the best results for recall, TSR, TP and FP. Romp has lowest specificity and precision values (0.671 and 0.719, respectively) since it reports a large number of FN and FP results (26 and 52, respectively). After some investigation, we found that Romp is built based on dyninst which is a binary instrumentation tool. It has the highest TSR, but it reports a lot of FP and FN test results. Another reason is DRB v1.3.0 adds more test cases using new OpenMP 5.0 features and GPU target offloading. Romp doesn't support those new features.

For RDS that is configured to use different policies, the union policy generates the highest recall value. However, this policy has the lowest accuracy since some underneath microservices (e.g. Intel Inspector and ThreadSanitizer) provide many FP results, which taint the final results. The intersection policy delivers the perfect score (1.0) for both specificity and precision. It indicates that for data race detection, the most conservative aggregation policy pays off in terms of reducing FP reports (all the way down to 0) and increasing TN (the highest value of 57). Another interesting observation is that the naive majority vote policy may not always beat the best performing individual tool in terms of accuracy and adjusted F1 score, due to the fact that the minority holds the truth sometimes. When there is an even number of microservices and a tie situation from individual tools, how to break the tie also has a significant impact on the final results as shown in the table 6. RDS using the directive-specific weighted vote policy achieves the best overall F1 score of 0.86. Compared to the best individual tool (Intel Inspector with 0.791), it is a relative improvement of 8.8%. And the directive-specific weighted vote policy is the best policy for RDS. Thus, it is used as the default policy for RDS. On average, RDS improves 12.6% Adjusted F-1 score, compared to the average Adjusted F-1 score (0.764) for the four individual tools.

It is also noticeable that RDS has the highest TSR number (97.9%) since it works as long as at least one tool supports a given test input file. Romp, which has the highest TSR among four tools, supports 93.8% tests in DataRaceBench.

4.4 Overhead

For RDS's metaservice, we measured the time from the curl command REST request to the end of the response. The total time includes the communication time between metaservice and microservices, the file transfer time between metaservice and microservices, the execution time of microservices, and the time for aggregating reports from multiple microservices. RDS takes about 13.07 hours to fully analyze DataRaceBench. The execution time of its result aggregator is 0.19 seconds, which is only 0.0038% of the total runtime of metaservice. Therefore the time overhead of metaservice's result aggregator can be ignored.

For RDS's microservices, the execution time is spent on running tools, using the result parser to analyze the results and generate detection reports in the JSON format we define, and finally aggregating reports from multiple runs. We measured the total execution time of both microservices and native tools using DataRaceBench.

Tool	Aggregate Policy	TP	FP	TN	FN	Recall	Specificity	Precision	Accuracy	TSR	Adjusted F1
Intel Inspector	Union	126	10	144	36	0.778	0.935	0.926	0.854	0.935	**0.791**
ThreadSanitizer		117	**1**	**151**	29	0.801	**0.993**	**0.991**	**0.899**	0.882	0.781
Archer		116	**1**	143	29	0.800	**0.993**	**0.991**	0.896	0.855	0.757
ROMP		**133**	52	106	**26**	**0.836**	0.671	0.719	0.754	**0.938**	0.725
RDS	Union	**144**	71	97	**19**	**0.883**	0.577	0.700	0.728	**0.979**	0.746
	Intersection	115	**3**	**165**	48	0.706	**0.982**	0.975	0.846	**0.979**	0.802
	Random	133	20	148	30	0.816	0.881	0.869	0.849	**0.979**	0.824
	Majority (Positive tie breaker)	132	9	159	31	0.810	0.946	0.879	0.868	**0.979**	0.850
	Majority (Negative tie breaker)	126	**3**	**165**	37	0.773	**0.982**	0.977	0.875	**0.979**	0.845
	Weighted Vote	132	8	160	31	0.810	0.952	0.943	0.882	**0.979**	0.853
	Directive-Specific Weighted Vote	130	**3**	**165**	33	0.798	**0.982**	**0.977**	**0.891**	**0.979**	**0.860**

Table 6: Quality of Individual Data Race Detection Tools and RaceDetectionService. DataRaceBench is used for generating the results. For 172 C tests, 83 of them have data races and 89 tests do not have data races. For 166 Fortran test cases, exactly half of them have data races and the other half are data race-free.

Thus, the time overhead of microservices can be calculated as $microService\_time - native\_time$.

As Table 7 shows, the overhead for microservices varies from one tool to another. The microservices wrapping Intel Inspector and ThreadSanitizer have less than 1 percent overhead, while the microservices for Archer and ROMP incur 4.54% and 7.05% overhead, respectively. Intel Inspector has high overhead because we configured it to use maximal resources to generate best results. Similar to a previous study [24], we also find that ThreadSanitizer has high overhead. The reason is that ThreadSanitizer isn't aware of OpenMP synchronization points. RDS's metaservice overall waiting time depends on the slowest performing microservice. We conclude that the overhead of using RDS is about 7%, mostly caused by the microservice layer.

Tools	Microservice	Native Use	Overhead	Extra Storage
Intel Inspector	15205 s	15129 s (99.5%)	76 s (0.50%)	90MB
ThreadSanitizer	13086 s	13015 s (99.45%)	71 s (0.55%)	90MB
Archer	1533 s	1467 s (95.46%)	66 s (4.54%)	90MB
ROMP	1034 s	966 s (92.95%)	68 s (7.05%)	90MB

Table 7: Time and Storage Overhead Comparison. The overhead indicates the time cost of parsing and transmitting result.

Table 7 also lists the extra RDS storage usage caused by using docker for each microservice. The storage usage of each microservice comes from the docker image and docker container. The image for Intel Inspector takes a lot of storage because the installation of the whole Intel package takes about 17GB. However, the storage overhead for the docker images, container and system files is 90 MB. The metaservice container uses 238MB disk space. Combined with the microservice layer's storage costs mentioned above, the final total storage overhead for RDS is about 598MB, which is 1.84% of the total docker image size of 32.58 GB.

4.5 NAS Parallel Benchmarks V3.0

We use NAS Parallel Benchmarks(NPB) V3.0 OpenMP C version[1] to evaluate RDS's capability of analyzing multiple-file packages. For multiple-file packages, RDS expects users to prepare a makefile with proper file dependencies and compiler flags. We have modified NPB's makefile (shown in Figure4) to be compatible with RDS. The S class of NPB programs is used for our experiment.

```
CC=clang #Define C compiler
CLINK=clang #Define Links C program
C_LIB=-L/usr/local/lib #Define link library
C_INC=-I../common -I/usr/local/lib #Define include
CFLAGS =-fopenmp -fsanitize=thread #Define compiler flags
CLINKFLAGS=-fopenmp -fsanitize=thread -O0 -lm -Wl,-rpath,/
     usr/local/lib #Define link time flags
UCC=clang #Define C compiler used to compile C utilities
BINDIR=../test #Define destination of executable binary file
```

Figure 4: Example configurations in a makefile

We use a Clang-plugin named OMPExtractor [26] to extract loop information in NPB programs and assign unique IDs to them. NPB has 730 loops in total. As a well-studied benchmark suite with built-in correctness verification, NPB programs do not contain data races. If a tool reports a data race for a given loop, it is considered as a false positive. Otherwise, the result is counted as a true negative. If a tool reports a data race's line information, we map the line number to the loop ID and consider it has a data race. Otherwise, we consider it a report of a data race. If a tool fails to report the data race location information, we consider the tool does not support the test. If a tool fails to process a program, we record one of the error codes such as compile-time segmentation fault (CSF), unsupported feature by a compiler (CUN), runtime segmentation fault (RSF) or runtime timeout (RTO).

Table 8 shows the results of using RDS to detect data races in eight NPB programs, ranging from BT to SP. Please note that F-1 score cannot be calculated since NPB has zero true positives and we cannot divide a number by zero. Therefore, we use Adjusted Accuracy instead.

The results show that ThreadSanitizer and ROMP had some compile or runtime errors, and they didn't report the exact data race location in the CG and LU program. Thus, their tools support rates are lower than those of Archer and Inter Inspector. Intel Inspector generated quite some false positives. Archer generated

Tool	Aggregate Policy	TP	FP	TN	FN	Accuracy	TSR	Adj. Accuracy
Intel Inspector	Union	0	100	630	0	0.863	**100%**	0.863
Thread Sanitizer		**0**	0	**649**	0	**1.000**	88.9%	0.889
Archer		0	1	630	0	**0.998**	**100%**	**0.998**
ROMP		0	**0**	151	0	**1.000**	20.6%	0.210
RDS	Union	0	100	630	0	0.863	100%	0.863
	Intersection	0	**1**	**729**	0	**0.998**	100%	**0.998**
	Random	0	22	708	0	0.970	100%	0.970
	Majority (Positive tie breaker)	0	**1**	**729**	0	**0.998**	100%	**0.998**
	Majority (Negative tie breaker)	0	11	719	0	0.985	100%	0.985
	Weighted Vote	0	**1**	**729**	0	**0.998**	100%	**0.998**

Table 8: Results of individual tools and RDS processing 730 loops in NPB's eight programs. All loops are free from data races.

the best results: 100% Test Support Ratio (TSR) and 99.8% adjusted accuracy (TSR*Accuracy) on NPB.

We investigated two benchmarks (MG and LU) further. For MG, Intel Inspector reports that function *psinv* and *resid* have data races. However, our investigation confirms that there are no data races in those functions. For LU, two tools (Intel Inspector and Archer) can analyze it without crashes or timeouts. Both tools report that a function named *blts* has data races. Our investigation finds that *blts* has two parallel loops using *nowait*, combined with a *flush* directive to synchronize memory operations. None of the two tools can recognize this complicated code pattern.

For RDS, the intersection policy, Majority vote and Directive Specific Weighted vote generate the best result (99.8% adjusted accuracy). All variants of RDS achieve 100% TSR. Since Archer generates high-quality results (almost perfect), RDS can not improve the results further compared to the best tool. We compared the average adjusted accuracy with four individual tools. On average, RDS improves 35% accuracy compared to the 0.739 average Adjusted Accuracy for four individual tools).

5 RELATED WORK

We categorize relevant work into two subareas in this section. However, we found no related services that support race detection of parallel programs.

5.1 Cloud Services and Aggregating Tools for Software Testing and Security

Software testing and security communities have adopted the idea of aggregating multiple tools for program analysis, testing and conformance testing. Code Dx application vulnerability management system [3] is a hybrid analysis and vulnerability scanner which combines and correlates the results generated by a wide variety of static and dynamic testing tools. It aggregates results from tools that support Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), and Interactive Application Security Testing (IAST) standards.

The Software Assurance Marketplace (SWAMP) [6, 19] provides both software and service to perform static code analysis on software source code. SWAMP provides the MIR-SWAMP [32] cloud computing platform to allow users to perform free code analysis online, and the standalone software application (SWAMP-in-a-Box) which can be deployed as a local instance of SWAMP. SWAMP provides APIs for direct access from (IDEs) such as Eclipse, source code management systems such as git and Subversion, and continuous integration systems such as Jenkins.

The Source Code Analysis Laboratory (SCALe) [31] is a static analysis aggregator and correlator which enables a source code analyst to combine static analysis results from multiple tools into one interface, and also provides mappings for diagnostics from the tools to the SEI CERT Secure Coding standards for assessing the conformance with the CERT standards. SCALe can be deployed on a web server for a web application and also is containerized as a standalone application.

5.2 Cloud Service for Programming Tools and Parallel Computing, and Containerized Software Platforms

Jupyter notebook framework [17] [15] supports creation of documents that have advanced computation and data processing capability for a domain. These documents can be considered as a web-based application for users since it allows programming, data processing, computation and visualization in a notebook cell as part of a document. For example, a Jupyter notebook can be set up as the front-end of parallel computing resources for users to write parallel programs in MPI [4].

Because of the complicated software environment for HPC, the DoE Extreme-scale Scientific Software Stack (E4S) [7] project uses Spack as the meta-build tool for a large collection of software and tools for HPC and provides containers of pre-built binaries for Docker, Singularity, Shifter, CharlieCloud, and so on. Our approach of creating microservices goes beyond the containerized software package by extending that to create virtualized services when the package is being deployed via docker-in-docker framework.

6 CONCLUSION

In this paper, we present RaceDetectionService, a novel approach to detect data race bugs in OpenMP programs. RDS uses a service-oriented design to provide a cloud-based race detection metaservice for users. Internally, it uses different policies to compose multiple microservices built on top of several individual data race detection tools using containers and REST API. We find that the Directive-Specific Weighted Vote is the best policy. The resulting framework

can leverage strengths of multiple tools while avoiding their individual limitations, subsequently generating better data race detection results. It also significantly improves the productivity of users. The framework is extensible to use a wide range of aggregate policies to merge conflicting results from multiple sources.

To the best of our knowledge, this work is the first effort in the HPC community to create cloud-based metaservice tools that consolidate other tools. It includes a service-oriented metaservice architecture design and implementation, the definition of common APIs and data exchange formats in the domain of data race detection, and several policies to compose potentially conflicting results from multiple tools. The same approach and framework can be used to create metaservice for compilers, performance tools, auto-tuning tools, etc. For policies to consolidate results from multiple tools, we demonstrate that policies incorporated with fine-granularity domain information such as OpenMP directive types deliver more accurate results than standard policies in our experiments. The results clearly prove that combining tools will indeed lead to much better results when the right aggregate policies are used.

In the future, we would like to compose supportive program analyses from different compilers and tools to have a more powerful data race detection tool. We also plan to support package management systems such as SPACK [10], in order to process more complex scientific applications that depend on third-party libraries.

ACKNOWLEDGEMENT

Prepared by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-809187) and supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research. This work is also supported by the National Science Foundation under the award 2015254.

REFERENCES

[1] [n.d.]. NAS Parallel Benchmarks 3.0. https://github.com/benchmark-subsetting/NPB3.0-omp-C.
[2] [n.d.]. Amazon Elastic Container Service Developer Guide. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-auto-scaling.html.
[3] [n.d.]. Code Dx. https://codedx.com/
[4] [n.d.]. Parallel programming with Jupyter. https://curc.readthedocs.io/en/latest/gateways/parallel-programming-jupyter.html.
[5] [n.d.]. Run the Docker daemon as a non-root user (Rootless mode). https://docs.docker.com/engine/security/rootless/.
[6] [n.d.]. Software Assurance Marketplace. https://continuousassurance.org/
[7] [n.d.]. The Extreme-scale Scientific Software Stack (E4S). https://e4s-project.github.io.
[8] 2020. Intel® Inspector. https://software.intel.com/en-us/inspector
[9] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamarić, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 53–62. https://doi.org/10.1109/IPDPS.2016.68
[10] Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R de Supinski, and Scott Futral. 2015. The Spack package manager: bringing order to HPC software chaos. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.
[11] Jorge Gomes, Emanuele Bagnaschi, Isabel Campos, Mario David, Luís Alves, Joao Martins, Joao Pina, Alvaro López-García, and Pablo Orviz. 2018. Enabling rootless Linux Containers in multi-user environments: the udocker tool. *Computer Physics Communications* 232 (2018), 84–97.
[12] Y. Gu and J. Mellor-Crummey. 2018. Dynamic Data Race Detection for OpenMP Programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 767–778.
[13] Sorin Lucian Lerner Jan Wen Voung, Ranjit Jhala. 2007. RELAY: static race detection on millions of lines of code. In *ESEC/FSE07: Joint 11th European Software Engineering Conference*. 205–214.
[14] Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. 1999. The OpenMP implementation of NAS parallel benchmarks and its performance. (1999).
[15] Jupyter. [n.d.]. Juputer Notebook. https://jupyter.org/.
[16] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and Accurate Static Data-Race Detection for Concurrent Programs. In *Computer Aided Verification*, Werner Damm and Holger Hermanns (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 226–239.
[17] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.
[18] Timur Iskhodzhanov Konstantin Serebryany. 2009. ThreadSanitizer: data race detection in practice. In *WBIA '09: Workshop on Binary Instrumentation and Applications*. 62–71.
[19] J. A. Kupsch, B. P. Miller, V. Basupalli, and J. Burger. 2017. From continuous integration to continuous assurance. In *2017 IEEE 28th Annual Software Technology Conference (STC)*. 1–8.
[20] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.
[21] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. ACM, New York, NY, USA, Article 11, 14 pages. https://doi.org/10.1145/3126908.3126958
[22] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[23] Chunhua Liao, Pei-Hung Lin, Markus Schordan, and Ian Karlin. 2018. A semantics-driven approach to improving DataRaceBench's OpenMP standard coverage. In *International Workshop on OpenMP*. Springer, 189–202.
[24] Pei-Hung Lin, Chunhua Liao, Markus Schordan, and Ian Karlin. 2018. Runtime and Memory Evaluation of Data Race Detection Tools. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 179–196.
[25] Pei-Hung Lin, Chunhua Liao, Markus Schordan, and Ian Karlin. 2019. Exploring Regression of Data Race Detection Tools Using DataRaceBench. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 11–18.
[26] Gleison Souza Diniz Mendonça, Chunhua Liao, and Fernando Magno Quintão Pereira. 2020. AutoParBench: a unified test framework for OpenMP-based parallelizers. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–10.
[27] Robert O'callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 167–178.
[28] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2006. LOCKSMITH: context-sensitive correlation analysis for race detection. *Acm Sigplan Notices* 41, 6 (2006), 320–331.
[29] Joachim Protze, Jonas Hahnfeld, Dong H Ahn, Martin Schulz, and Matthias S Müller. 2017. OpenMP tools interface: Synchronization information for data race detection. In *International Workshop on OpenMP*. Springer, 249–265.
[30] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.
[31] Robert Seacord, Will Dormann, James McCurley, Philip Miller, Robert Stoddard, David Svoboda, and Jefferson Welch. 2012. *Source Code Analysis Laboratory (SCALe)*. Technical Report CMU/SEI-2012-TN-013. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=10091
[32] SWAMP. [n.d.]. Software Assurance Marketplace (SWAMP) code-analysis service. https://www.mir-swamp.org/.
[33] Gaurav Verma, Yaying Shi, Chunhua Liao, Barbara Chapman, and Yonghong Yan. 2020. Enhancing DataRaceBench for Evaluating Data Race Detection Tools. In *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 20–30.
[34] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 221–234.