SERG: A Static Data Race Detector for Exo Programs

Andrew Alex
adalex@cs.washington.edu
University of Washington

Anjali Pal
anjalip@cs.washington.edu
University of Washington

Ido Avnon
idoavnon@cs.washington.edu
University of Washington

1 Introduction

User-scheduled languages (USLs) are an emerging class of programming languages built to explicitly express performance optimizations. The power of a USL comes from the division of a program into two pieces: an algorithm or object program that specifies the program's *functional* behavior, and a schedule that defines *how* the computation is performed, by transforming the original, naïve algorithm into a high-performance version. USLs typically provide guarantees that scheduling transformations preserve functional equivalence to the original algorithm.

State-of-the-art USLs work best on small, computationally intensive kernels with a high degree of data parallelism. In such cases, program schedules combine SIMD hardware primitives with a variety of loop transformations to yield highly efficient programs. Unfortunately, existing user-scheduled languages, such as Halide [8], Elevate [5], and Exo [6], have limited support for kernels that execute concurrently, or the concurrency is not user-scheduled and is entirely managed by the compiler. User-scheduling concurrency is difficult, in part, because of the potential to introduce arbitrary data races into the object program, thus making it difficult to verify against the original sequential program. A key feature of Exo is that it *externalizes* compiler backends; that is, algorithms like instruction selection are not automated and instead performed by the programmer via the scheduling language, then verified for functional equivalence against a specification of the instruction that exists outside the compiler. This externalization allows performance-concerned programmers to optimize programs targeting novel hardware that would otherwise require larger-scale compiler modifications, including optimization of concurrent programs. In order to support user scheduling of concurrent kernels, we have extended Exo experimentally with `fork` and `barrier` primitives for creating and synchronizing threads, respectively. Since this is an experimental extension for the purposes of this project, there are some limitations in the programming model compared to a full-featured fork and barrier library. In particular, nested fork bodies are not permitted and there is only one global barrier that all forked threads are subscribed to and must wait at before any can proceed.

We introduce the Static Exo Race Guardian (SERG), a static analysis tool to detect data races in this experimental extension of Exo. The focus of our tool is not on the actual introduction of the concurrency via a schedule, but rather verifying the final, scheduled program is free of data-races. Scheduling will be handled outside the scope of this class project.

2 Example

As a concrete example of the problem we are solving, consider the 1D stencil computation illustrated in Figure 1. Each grid represents the same underlying array at different time steps. Each location is updated by a weighted combination of its neighbors from the previous time step. After writing an Exo program that defines this

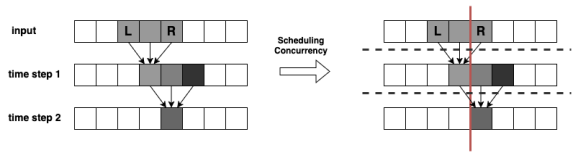

Figure 1: 1-D stencil computation before and after the data is divided between threads (red, vertical line) and barriers (horizontal, dashed line) are inserted for synchronization. Only single cell computation shown. Barriers are needed when L or R crosses the division of the buffer. Time proceeds down after each element has been computed.

functional description of the algorithm, the programmer may wish to optimize the program by scheduling in concurrency. In order to accomplish this optimization, suppose that the schedule divides the data in half and computes with two threads. To preserve program correctness, the reads of elements bordering division must happen after the update by the neighboring thread in the previous timestep. This necessary synchronization is indicated by the horizontal, dashed line in the diagram on the right. Without the barrier, the read and write to the same location may be reordered and thus introduce a data race. Specifically, a data race exists if there is a write to a location by one thread and a read or write to the same location by a different thread not separated by a barrier. Unfortunately, there does not exist a mechanism to detect this conflict in Exo programs, which is necessary to preserve Exo's guarantee of safe scheduling transformations.

3 Technique

Static data race detection for general-purpose languages is challenging, in part, because it relies on an accurate pointer analysis. Computing a fully precise pointer analysis is undecidable, so existing static race detection algorithms must make approximations, which limit their effectiveness in practice. However, because Exo is a domain-specific language with limitations on the space of programs that may be expressed, our analysis will not have to handle details that make arbitrary data race detection difficult. Specifically, there are no pointers in Exo and only limited mechanisms for aliasing a data buffer (the state in the program that we are interested in detecting data races over), so we will be able to precisely determine what memory is accessed statically. Furthermore, Exo programs are largely *static-control programs*; that is, programs whose control flow can be statically determined and does not depend on program data or inputs. There are also no recursive function calls in Exo programs. There are limited exceptions to these constraints (e.g., control flow may depend on the length of the input buffers, which is itself an input), but the technique we outline is able to handle these exceptions. In other words, we can precisely compute an expression for all the accesses parameterized by thread IDs and loop bounds

and dispatch to an SMT solver to determine if there are conflicting accesses between barriers. The high-level algorithm for SERG is presented below.

Listing 1: SERG Algorithm

```
for each concurrent region:
  reads = {}
  writes = {}
  for t in body:
    reads[t] = construct_symbolic_reads(t)
    writes[t] = construct_symbolic_writes(t)

  for t1, t2 in threads:
    solve(writes(t1).intersect(
            reads(t2).union(writes(t2)))
          == empty)
```

Since threads in concurrent Exo programs are synchronized by barriers, we only need to consider potential data races within a region between barriers. For each region, we simply need to collect all of the reads and writes that occur, and construct a query to a solver that can check whether a data race is possible. The construction of these regions is outlined in section 4.2. In the following section, we outline how we can compute the symbolic expression for the accesses in the presence of various language features.

4 Implementation

SERG is available publicly at https://github.com/andrewdalex/exo-cse503. To implement SERG, we extended the Exo programming language with primitives to fork threads and wait at a barrier. We added support for these features to the front-end of the Exo compiler and extended the typechecker and transformation process to the Exo internal representation (IR), called *LoopIR*. This IR provides a thin layer over the abstract syntax tree and is the input to our analysis algorithm. A single pass through the LoopIR constructs the SMT expression for the read and write indices of all shared variables, which are more precisely all the variables in-scope prior to the execution of the fork statement. This set is parameterized by an index metavariable as well as the thread id parameter. At each access point a clause is added to the access set (via conjunction with prior accesses) of the following form:

$$access = i \wedge i = \text{dom(i)} \wedge \text{control\_refinement}$$

where `dom(i)` gives all values i is allowed to take on in the fork body and `control_refinement` is a restriction on *when* this access may be executed.

4.1 Query Construction Example

To illustrate a concrete example of this access set contruction, consider the following multi-threaded Exo program:

```
@proc
def racey_parallel_memzero(n: size, A: i32[n]):
  for thread_id in fork(2):
    for i in seq(0, n-1):
      A[i + thread_id] = 0
```

This program contains a data race as it zeroes a buffer `A` with two threads. Note, that the "`for thread_id in fork`" statement is **not** a loop and is just the syntax for spawning two threads. There is only one program point where `A` is accessed so the access expression has only one clause. Namely, the following:

$$(access = i + thread\_id) \wedge (0 \leq i < n - 1) \wedge \texttt{true}$$

That last clause (`true`) captures the control flow refinement— in this case, no refinement is needed because the access is not under a branch. All variables involved in the indexing expression have their domains added to the domain sub-clause except thread ID parameters which are treated as a special case.

When the fork body joins, we loop over all shared-variables with accesses in the fork body and compute the set intersection. Before issuing the query, we perform variable substitution in the equation so that they may vary independently in each access set from one another (e.g. we want to allow `i` to take on different values in each set). The `access` and `thread_id` variables are also substituted with the new `thread_id` allowed to take on all possible values. We then ask the solver if $access_1 = access_2$, with the subscripts representing the substituted values on each side of the equation. The final query looks like this:

$$\begin{gathered}((access_1 = i_1 + thread\_id_1) \wedge (0 \leq i_1 < n - 1) \wedge \texttt{true}) \\ \wedge \\ ((access_2 = i_2 + thread\_id_2) \wedge (0 \leq i_2 < n - 1) \wedge \texttt{true}) \\ \wedge \\ (0 \leq thread\_id_1, thread\_id_2 < 2) \wedge thread\_id_1 \neq thread\_id_2 \\ \wedge \\ access_1 = access_2\end{gathered}$$

SERG constructs this query for both the write access set and the write/read access sets. If the solver can satisfy this query, then there is a data race. SERG dispatches the query to the Z3 SMT solver [3] using the theory of bitvectors via the pySMT library [4].

4.2 Region Splitting

Our implementation splits the input program into concurrent regions, which are independently verified. In this context, a concurrent region is a sequence of statements executed by multiple threads, but where no synchronization is present. In other words, all accesses in a concurrent region may be interleaved between threads arbitrarily. Additionally, accesses deterministically belong only to one concurrent region.

There are 3 different ways to construct a concurrent region. The first, and simplest, is a fork body without barriers present – the entire fork body is the concurrent region as there is no synchronization within the body. The second case is a barrier within a sequence of straight-line code. This barrier splits the block of code into two regions: one containing the statements before the barrier and one consisting of the statements after the barrier. Finally, the most complicated case is a barrier within a loop, which creates 3 regions. The first region contains all the code in the region before the loop and the statements in the loop up to the barrier. The second region includes the statements after the barrier as well as the statements before the barrier. Note that in this case, we have to reflect the different values for the loop index variable within the region. The third region contains all the statements after the barrier in the loop as well as the statements after the loop up to

the next synchronization point. In the case where multiple barriers are present in the loop, an additional region is created for each barrier containing the statements which do not span multiple loop iterations (i.e., the statements between barriers where the value of the loop index is effectively fixed for all threads.) In each of these cases, we have to modify our query to use the correct value for the loop index variable.

4.3 Additional Program Patterns

In this section, we detail how other Exo program constructs are handled in SERG with particular attention to those that are helpful to write useful, parallel programs. While we are still in the process of extending the features handled, we are currently able to verify data race-freedom on an expressive enough subset to translate several microbenchmarks of interest, detailed in Section 5.

Loops outside of a Fork. This is a useful pattern to spawn threads for every timestep of a big computation, for example. If you look at our example query in the last section, you will notice that the loop iterator variables were allowed to vary independently, which should not be the case if the fork is instead inside the loop. The solution to this is actually more general: SERG does not perform the symbol substitution on variables allocated before the fork starts. This effectively "captures" a single value of each shared variable across all threads, though that value is still allowed to vary across its domain. In other words, every thread sees the same value, but SERG checks all possible values.

If Statements. Conditional branches modify the control clause of the accesses nested underneath them with the condition they specify (including the negation of the condition for else clauses).

Process Preconditions. A key feature of Exo is the ability to specify preconditions on the input via assert statements, which help many of the safety checks complete. SERG handles preconditions by adding them to the control refinement clause of all accesses.

Barriers. Barriers form the concurrent regions we analyze, as discussed in the previous section. Our implementation requires that barriers are never conditionally executed (i.e., they are not under an if statement). However, because Exo has static control, this requirement is just for engineering purposes and does not pose theoretical restrictions to our verification. However, a barrier under a branch introduces the possibility of deadlock, which is complexity we chose not to handle.

Windowing, Multi-Dimensional Buffers, and Function Calls. We currently do not handle windowing and also assume that all buffers have one indexing dimension (without loss of expressivity). We also do not handle procedure calls yet, but they will be trivially enabled with windowing as the call arguments can just be interpreted as windows to the underlying variables in the caller and the procedure body inlined. This inlining strategy is possible in Exo because there is no recursion.

5 Evaluation

Our evaluation aims to answer the following research questions:

RQ1 How well can SERG identify data races in concurrent Exo programs?

RQ2 Does SERG perform better than state-of-the-art race detectors operating on the same programs written in C?

In order to answer these questions, we initally planned on using DataRaceBench [7], a micro-benchmark suite for data race detection. After translating three of the benchmarks to Exo (see DRB001, DRB112, and DRB120 in [7]), we realized that DataRaceBench is a poor benchmark suite for our evaluation. Primarily, this is due to the fact that most of the tests exploit features in C that are not expressible in Exo. Consider the following test, labeled DRB195:

```
double *u1, *u2, c = 0.2;
int n = 10, nsteps = 10;

int main()
{
  u1 = malloc(n * sizeof(double));
  u2 = malloc(n * sizeof(double));
  for (int i = 1; i < n - 1; i++)
    u2[i] = u1[i] = 1.0 * rand() / RAND_MAX;
  u1[0] = u1[n - 1] = u2[0] = u2[n - 1] = 0.5;
  for (int t = 0; t < nsteps; t++)
  {
#pragma omp parallel for
    for (int i = 1; i < n - 1; i++)
    {
      u2[i] = u1[i] + c * (u1[i - 1] + u1[i + 1] - 2 * u1[i]);
    }
    double *tmp = u1;
    u1 = u2;  // u2 = tmp;
  }
  for (int i = 0; i < n; i++)
    printf("%1.2lf ", u1[i]);
  printf("\\n");
  free(u1);
  free(u2);
}
```

This program is not expressible in Exo due to the highlighted statement, which aliases two pointers. Many of the inputs in DataRaceBench are not portable to Exo for similar reasons. With this realization, we focused on writing tests directly Exo that demonstrate the capabilities of SERG. For example, the following test demonstrates that SERG properly tracks restrictions on array access locations by branch conditions:

```
def if_else_safe(a: i8[10]):
        for tid in fork(2):
            if tid == 0:
                for i in seq(0, 5):
                    a[i] = 0
            else:
                for i in seq(5, 10):
                    a[i] = 1
```

All of the tests, including those ported from DataRaceBench, are summarized below.

Table 1: Results from running SERG on our test suite and 3 DataRaceBench micro-benchmarks.

Test Name	Data Race? (Y/N)	Result
drb001_yes	Y	TP
drb112_no	N	TN
drb120_no	N	TN
detect_trivial_dr	Y	TP
verifies_trivial_safe	N	TN
verifies_trivial_loop_unsafe	Y	TP
verifies_overlapping_zero_copy_unsafe	Y	TP
reads_overlap_writes	Y	TP
reads_not_overlap_writes	Y	TP
thread_local_ok	N	TN
basic_branch_unsafe	Y	TP
basic_branch_safe	N	TN
if_else_safe	N	TN
if_else_unsafe	Y	TP
array_dependent_type_restricts_racey_loop	N	TN
fork_in_loop_safe	N	TN
fork_in_loop_unsafe	Y	TP
barrier_simple_safe	N	TN
barrier_simple_unsafe_before	Y	TP
barrier_simple_unsafe_after	Y	TP
barrier_three_regions_safe	N	TN
barrier_three_regions_unsafe	Y	TP
barrier_three_regions_unsafe	Y	TP
multiple_forks_safe	N	TN
multiple_forks_unsafe_first	Y	TP
multiple_forks_unsafe_second	Y	TP
verifies_loop_split_zero_copy_safe	N	TN
single_barrier_in_loop	N	TN
pathalogical_looping_no	N	TN
pathalogical_looping_yes	Y	TP
triple_nested	N	TN

Note that SERG is a sound analysis while the other tools evaluated are not. However, on the three input programs ported from DataRaceBench, the other race detection tools also produce the correct results. All experiments were run on the University of Washington's CSE Instructional Linux Cluster, which operate on SuperMicro, 192GB RAM, 2 14-core Intel Gold 6132 processors @ 2.6GHz.

6 Related Work

Deterministic Parallel Java (DPJ). DPJ [2] is an extension to the Java type system, which enforces determinism (and thus race-freedom) in multi-threaded programs as a compile-time guarantee. DPJ unfortunately relies on programmer-provided region annotations to enforce separation between objects on the heap; however, it is able to provide guarantees about a broader class of programs than can be expressed in Exo. Our mechanism for tracking accesses through windows on buffers is also more powerful than DPJ's sub-array type, which does not support strided access patterns.

Compositional Static Race Detection RacerD [1] is a static program analysis built for speed and scale in Java programs in the context of software engineering. RacerD is motivated by the integration of parallelism into Facebook's News Feed, which was used as an evaluation metric. RacerD utilizes an abstract domain for heap manipulation in Java, incorporating multiple lock ownership, distinction between main and non-main threads, and use of the `synchronized` keyword in the language. Its scale and speed within a modern software engineering context relies on its use in deployment. It is run during code review on diffs – increasing usability as developers are already within a debugging context. RacerD also has different priorities as an analysis, "favoring reduction of false positives over false negatives," reinforcing the idea that false positives reduce confidence for developers. RacerD, like SERG, is a powerful static analysis for race detection. However, it must make compromises in the soundness of its analysis due to the scale of its language and usage. SERG, operating on a small scale in Exo, is able to produce a more precise analysis.

Accurate Static Data Race Detection for C. CSeq-DR [9] is a static race detection tool for C programs. It shows the reduction of the problem of race detection to a reachability analysis. They augment programs with auxiliary global variables tracking the target address and length of shared memory locations and the thread id of the thread performing the operation. They further augment the program with guarded assertions over these auxiliary variables before reads and writes to shared data such that a violation of any assertion indicates a data race. The augmented program is then checked via bounded model checking to ensure that none of the assertions can be violated up to a set bound, thus ensuring there are no data races up to that bound. CSeq-DR is able to achieve highly accurate data race detection with no reported false positives. However, since it relies on bounded model checking, it could miss data race bugs if the bounds are too small. SERG makes the opposite trade-off– it is sound, so it will never miss a potential data race, but precision may suffer as a result.

References

[1] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. In *Proceedings of the ACM on Programming Languages*. Association for Computing Machinery, New York, NY, USA, 1–28. doi:10.1145/3276514

[2] Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 97–116. doi:10.1145/1640089.1640097

[3] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[4] Marco Gario and Andrea Micheli. 2015. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop 2015*.

[5] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. 2020. A Language for Describing Optimization Strategies. *CoRR* abs/2002.02268 (2020). arXiv:2002.02268 https://arxiv.org/abs/2002.02268

[6] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. doi:10.1145/3519939.3523446

[7] 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)*. Association for Computing Machinery, New York, NY, USA, Article 11, 14 pages. doi:10.1145/3126908.3126958

[8] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association

for Computing Machinery, New York, NY, USA, 519–530. doi:10.1145/2491956.2462176

[9] Emerson Sales, Omar Inverso, and Emilio Tuosto. 2025. Accurate Static Data Race Detection for C. In *Formal Methods*, André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi (Eds.). Springer Nature Switzerland, Cham, 443–462.