Taskgrind: Heavyweight Dynamic Binary Instrumentation for Parallel Programs Analysis

Romain Pereira, George Stelle, Patrick Carribault

▶ **To cite this version:**

Romain Pereira, George Stelle, Patrick Carribault. Taskgrind: Heavyweight Dynamic Binary Instrumentation for Parallel Programs Analysis. 8th International Workshop on Software Correctness for HPC Applications (Correctness '24), Nov 2024, Atlanta (USA), United States. hal-04814885

HAL Id: hal-04814885
https://hal.science/hal-04814885v1

Submitted on 2 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Taskgrind: Heavyweight Dynamic Binary Instrumentation for Parallel Programs Analysis

Romain PEREIRA
Avalon, LIP, ENS, Inria
Lyon, France
romain.pereira@inria.fr

George STELLE
Los Alamos National Laboratory
Los Alamos, New Mexico, United States
stelleg@lanl.gov

Patrick CARRIBAULT
CEA, DAM, DIFF-91297 Arpajon
Université Paris-Saclay, CEA,
Laboratoire en Informatique Haute Performance pour le Calcul et la Simulation
Bruyères-le-Châtel, France
patrick.carribault@cea.fr

***Abstract*—Determinacy races are concurrent programming hazards occurring when two accesses on the same memory address are not ordered, and at least one is writing. Their presence hints at a correctness error, particularly under asynchronous task-based parallel programming models. This paper introduces Taskgrind: a Valgrind tool for memory access analysis of parallel programming models such as Cilk or OpenMP. We illustrate the tool's capabilities with a determinacy-race analysis and confront it with state-of-the-art tools. Results show fewer false negatives and memory overheads on a set of microbenchmarks and LULESH, with meaningful error reports toward assisting programmers when parallelizing programs.**

***Index Terms*—HPC, Task, Binary Instrumentation, Determinacy Race**

I. INTRODUCTION

Determinacy races are concurrent programming hazards occurring when two accesses on the same memory address are not ordered, and at least one is writing [1]. The presence of determinacy races in a program can make its output vary given the same input, which may hint at the presence of correctness errors. With the ever-growing intra-node parallelism of supercomputers, porting simulation codes while ensuring parallel execution correctness is challenging. It is particularly true with dependent task-based programming models over more synchronous programming models. A missing synchronization can lead to an incorrect order of execution and, in the end, to an erroneous simulation.

To help prevent unintended determinacy races, many tools had been developed, which we summarize briefly here. LLOV [2] is a static data-race checker, meaning it detects the presence or absence of errors at compile-time, hence with a limited vision of the program. Archer [3] and TaskSanitizer [4] instruments at compile-time to perform run-time analysis falling back to LLVM' ThreadSanitizer [5]. Motivations and results of Archer [3] showed its effectiveness for *debugging* applications, even at large scale. However, as suggested by E. Dijkstra [6], *"debugging is an inadequate means for ensuring programs correctness and that we must prove the correctness of programs"*. In particular, both tools are subject to *false negative*: it may report no data race in an erroneous program. One source of false negatives comes from non-instrumented code - for instance, in vendor-specific dynamic libraries which source code may not be visible at compile-time. ROMP [7] is a static binary instrumentation tool relying on Dyninst [8] for checking OpenMP program's correctness. It tackles Archer/TaskSanitizer's limitation by recompiling and instrumenting *binary programs*. However, ROMP' main drawbacks are its poor error reporting and that it only supports OpenMP semantics.

This paper explores the use of heavyweight Dynamic Binary Instrumentation (DBI) towards (1) reducing false-negatives detection, (2) meaningful determinacy race reports and (3) multi-parallel programming model support. Our contributions are:

- the introduction of Taskgrind: a Valgrind tool for parallel programming model memory accesses analysis - with support for OpenMP and a work-in-progress Cilk support.
- a brief survey on pitfalls related to heavyweight DBI of parallel programs,
- the implementation of a determinacy race analysis in Taskgrind, and its evaluation with respect to state-of-the-art tools.

The paper is organized as follows. Section II provides relevant background. Section III introduces the design of Taskgrind and the determinacy race detector implementation. Section IV presents a few pitfalls on parallel programs heavyweight DBI, and how Taskgrind workarounds them. Section V evaluates Taskgrind overheads and its determinacy race detection analysis. Finally, Section VI review the literature, and we conclude in Section VII.

II. BACKGROUND

Here we introduce the segment graph data structure and Valgrind concepts on which rely Taskgrind.

A. Segment Graph

Parallel programs can be represented as a *segment graph* where nodes are non-divisible sequences of instructions, and so that there exists a path from nodes N_i to N_j if and only if a synchronization imposes N_i to *happen-before* N_j. Fig. 1 illustrates such a graph, used in race detector tools such as TaskSanitizer [4] or ROMP [7].

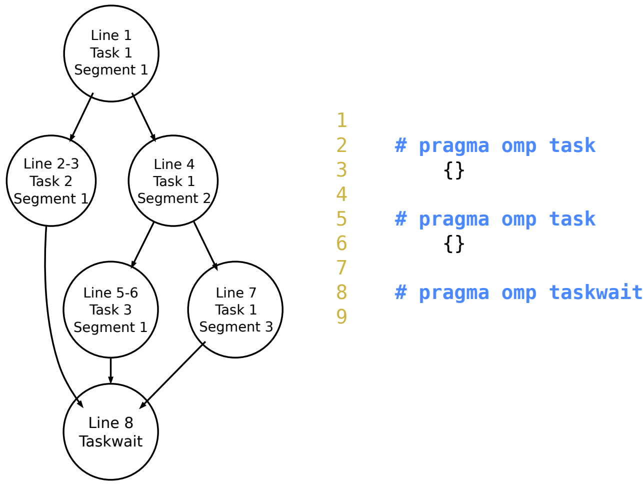


Fig. 1: Minimal example of a segment graph

To reason about segment graphs, we build on the formal definitions of TaskSanitizer [4]. In particular, we refine the definition of a happens-before relation (HB, $\prec$) to include reasoning about parallel regions. Precisely, if one parallel region happens before another, than all task segments in the first happen before all test segments in the second. Formally:

$$\frac{p1 \prec p2}{\forall t1 \in p1, \forall t2 \in p2, t1 \prec t2} \quad (1)$$

This is a useful definition in implementing and reasoning about OpenMP (and perhaps Qthreads), while Cilk programs can be assumed to have a single parallel region containing all tasks.

B. Valgrind

Valgrind is a Dynamic Binary Instrumentation (DBI) framework [9]. It performs just-in-time recompilation of code blocks from binary programs to the VEX intermediate representation (IR). A Valgrind *tool* includes the Valgrind *core* and a *plugin*. The IR tree is generated by the core and passed to the plugin that can inject IR instructions for instrumenting. In addition, Valgrind provides utilities such as *client requests* - so the instrumented program (aka. the client) can forward information to the tool - or *function replacement*, used, for instance, by the default tool memcheck to wrap memory allocators. Valgrind also makes read/write instruction instrumentation relatively simple through the VEX IR.

III. Taskgrind: A Tool for Parallel Program Memory Accesses Analysis

We present Taskgrind, a Valgrind tool designed to analyze parallel program memory accesses. Fig. 2 summarizes its design. The application may be any program supported by Valgrind. It must rely on a parallel programming model that provides a segment graph to Taskgrind through client requests. Section III-A presents the considered parallel programming models and their support, Section III-B the per-segment memory accesses recording, and finally Section III-C an analysis for determinacy race detection.

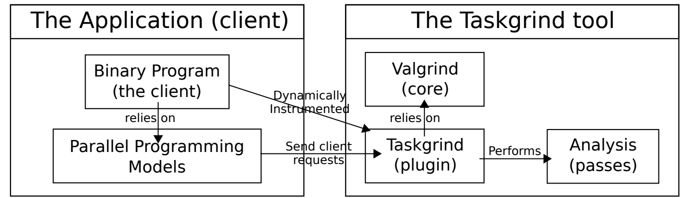


Fig. 2: The design of Taskgrind

A. Parallel Programming Models Support

In order to support a programming model, Taskgrind must build a segment tree from a program execution. We briefly introduced considered programming models and their current support.

a) OpenMP: OpenMP is a parallel programming model with a main focus on intra-node parallelism. It provides many synchronization mechanism such as threads fork-join, mutex, barriers, dependencies or detachable tasks. It also comes with a standard tooling interface: OMPT [10]. An OMPT-tool can register callbacks raised on specific run-time events. Taskgrind provides a built-in OMPT-tool that forward the OpenMP program state to the Taskgrind plugin via client requests. The OMPT-tool is automatically injected into the instrumented program by Taskgrind. Such approach to building a segment tree is similar to TaskSanitizer [4], the main differences stands in the supported interfaces: for instance, TaskSanitizer supports mutexes but does not support the `inoutset` dependency type nor the `detach` clause, while Taskgrind is the opposite.

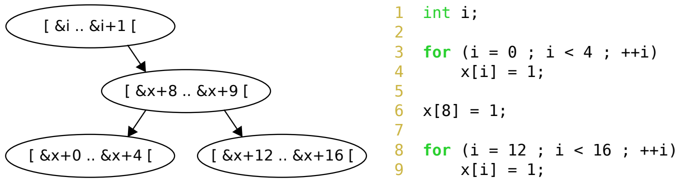
b) OpenCilk: Cilk is a programming model that uses tasks as its fundamental building block [11]. While there is a race detector for Cilk [12], it has the previously mentioned drawback of requiring compile-time instrumentation. The implementation of the Cilk runtime, Cheetah, takes a drastically different approach from OpenMP, making a Taskgrind integration difficult. As a result, we have a work-in-progress implementation[1].

c) Qthreads: Qthreads is a userspace threading runtime that implements tasks and other synchronization primitives [13]. It provides a wide range of primitives for organizing dependencies between tasks, which presents unique challenges for taskgrind. As a result, instrumenting qthreads for use with Taskgrind will require subtle extensions to Taskgrind semantics to allow handling of constructs such as the full/empty-bits (FEBs). That said, we hope to get some of the basic tasking semantics of Qthreads instrumented for use with Taskgrind.

B. Memory Accesses Recording

While building the segment tree, Taskgrind keeps track of the active segment on each thread. Two interval trees are attached to each segment to record *read* and *write* access.

[1]https://github.com/stelleg/cheetah/tree/taskgrind




```
1  int i;
2
3  for (i = 0 ; i < 4 ; ++i)
4      x[i] = 1;
5
6  x[8] = 1;
7
8  for (i = 12 ; i < 16 ; ++i)
9      x[i] = 1;
```

Fig. 3: Write interval tree for the right-side segment

Fig. 3. Such structure allows compactly accumulated dense memory accesses and a light $O(log\ n)$ complexity on most tree operations used in later analysis - n being the number of dense intervals accessed.

C. Determinacy Race Analysis

When the parallel program finishes execution, Taskgrind will have generated a segment tree representation of the execution, with accesses recorded for each segment. Using that knowledge, Algorithm 1 presents the implementation of a determinacy race analysis. It compares each independent-segments to one-another, intersecting their accesses location where at least one is writing. If the intersection is not empty, then a determinacy race may be reported.

Algorithm 1 Determinacy Race Analysis Pass

Require: Segment graph G
1: **for each** segment $s_1 \in G$ **do**
2:     **for each** segment $s_2 \in G \backslash \{s_1\}$ **do**
3:         **if** no path exists between s_1 and s_2 **then**
4:             **if** $s_1.w \cap (s_2.r \cup s_2.w)$ is not empty **then**
5:                 Possible determinacy races $(s_1,\ s_2)$

In order to provide meaningful report to the programmer, Taskgrind overloads the memory allocator using Valgrind interfaces, to save a stack trace on each memory block allocation, so conflicting accesses can be matched with source-code location reading debugging information if compiled-in the binary.

IV. Pitfalls with Heavyweight Instrumentation

The original motivations of Taskgrind were to avoid false-negative detection by instrumenting every instruction of the program understudy. However, naively running Taskgrind and its determinacy race analysis leads to many false positives. Instrumenting an OpenMP dependent task-based version of LULESH [2] reports about 400,000 determinacy races with a low problem size `-s 4` (the mesh size) and low parallelism `-tel 2` (the number of tasks per loop). This section presents a few sources of false-positives, illustrated with OpenMP, and how Taskgrind suppresses them.

A. Parallel Runtimes Non-Determinacy

Reporting every determinacy race is not always relevant for correctness analysis. In fact, correct parallel runtime systems (in the sense that they respect their specifications) can be non-deterministic themselves. For instance, OpenMP and Cilk implementations rely on work-stealing task scheduling: given the same tasks, different execution may lead to different scheduling.

In order to turn off the report of such unharmful determinacy races, we added two lists of symbols: the *ignore-list* and the *instrument-list*. The former one tells Taskgrind to turn off instrumentation for accesses occurring in a symbol matching any. The latter tells Taskgrind only to allow instrument accesses occurring in a symbol matching any. For instance, the *ignore-list* contains symbols prefixed with `__kmp` corresponding to LLVM' OpenMP runtime.

B. Memory Recycling

```
1  for (int i = 0 ; i < 2 ; i++)
2  {
3     # pragma omp task   // T(i)
4     {
5        void * x = malloc(4);
6        write(x);
7        free(x);
8     }
9  }
```

Listing 1: Memory Recycling false-positives

Another source of false positives stems from memory allocators. Listing 1 shows a minimal example where two tasks allocate memory, write on it, and deallocate it. While segments associated with OpenMP tasks $T(0)$ and $T(1)$ are declared independent, but run-time scheduling may lead the system allocator to allocate/deallocate the same memory location x by recycling memory. Taskgrind bypasses this issue by overloading the system allocator. It transforms memory deallocation calls such as `free` or `delete` to a `no-op`. This workaround ensures that two allocations will always point to distinct memory addresses, removing false positives from recycling.

It must be noted that Valgrind memory allocator overriding does not support custom memory allocators. For instance, the LLVM OpenMP runtime 19.x has its own memory allocator that can also perform memory recycling (`___kmp_fast_allocate`). Extending the support of memory allocators is kept as future work.

C. Thread-Local Accesses

```
1  _Thread_local int * x;
2  [...]
3  for (int i = 0 ; i < 2 ; i++)
4   # pragma omp task   // T(i)
5      write(x);
```

Listing 2: Memory Recycling false-positives

Thread-local accesses are also a source of false positives. For instance, it includes memory allocated via the `pthread_key_create` interface, C11 specifier `_Thread_local`, or OpenMP `threadprivate` directive. We focus on the support of the `_Thread_local` specifier, but concepts could be extended to other thread-local storage.

[2] https://github.com/rpereira-dev/LULESH

On the Listing 2, both $T(0)$ and $T(1)$ may be scheduled on the same thread and Taskgrind would record two independent tasks writing the same memory location. To suppress such false-positive, Taskgrind records the Thread Local Storage (TLS) [14] information of the executing thread. When a segment completes, the Thread Control Block (TCB) and the Dynamic Thread Vector (DTV) are saved and attached to the segment. In the determinacy race analysis, if both accesses occurred on the same thread and in the same DTV, then the report is suppressed.

However, this approach is limited. A segment could allocate a new DTV block, access it, and de-allocate it before the segment completion, which means it would not appear in the recorded DTV. Such a scenario would still lead to false-positive reports by the determinacy race analysis. Using TLS *gen* number, Taskgrind could detect a change of the DTV structure that occurred during the segment execution and warn the user, but a false-positive would still be reported. A better solution kept as future work is to use Valgrind function replacement for TLS allocation/de-allocation record TCB/DTV while splitting the current segment when it occurs.

Another limitation is *user-based* thread-local accesses. A pattern commonly found in OpenMP code is indexing an array with the executing thread - `array[omp_get_thread_num()]` - which is not detected currently by Taskgrind.

D. Segment-Local Accesses

```
1  for (int i = 0 ; i < 2 ; i++)
2     # pragma omp task    // T(i)
3       int x = 0;
```

Listing 3: Memory Recycling false-positives

The last source of false-positive suppression presented in this paper is depicted on Listing 3. Running with code with LLVM' OpenMP, both segments associated with tasks $T(0)$ and $T(1)$ may be executed on the same thread one after the other. With LLVM/GCC implementation, x is pushed at the same address onto the executing thread stack, resulting in a conflict when analyzing. In order to suppress such false positives, Taskgrind registers the stack frame address at the start of a segment. Then, during the determinacy analysis, such false positives are suppressed by confronting conflicting stack accesses with registered stack frames.

V. Evaluation

The original motivations of Taskgrind were to have no false negative reports (as compared to Archer and TaskSanitizer) - while improving error reports (as compared to ROMP). This section evaluates Taskgrind determinacy race analysis, its overhead over execution time, and memory use, and illustrates its error reporting capabilities[3]. Every program was compiled using LLVM and disabling optimizations (option `-O0`).

[3]Software versions and source code are referenced in the Appendix of this document.

A. Microbenchmarks

We evaluate Taskgrind determinacy race detection analysis on a subset of the DataRaceBenchmarks [15] focusing on task-related constructs. We extended it with seven Taskgrind-specific microbenchmarks (TMB) treating heavyweight DBI pitfalls discussed in Section IV. We run TMBs with 1 and 4 threads to:

- force memory recycling, thread-local accesses, and segment-local accesses of independent segments,
- ensures the tool captures the code semantic and not implementation specific behavior. For instance, LLVM implementation makes every task *included* when executing on a single-thread [4].

Results are depicted on Table I and confront state-of-the-art analysis tools: TaskSanitizer [4], Archer [3] and ROMP [7]. The first column is the benchmark name, the second is whether a data race is present, and the other columns depict reports obtained by the specified tools. Letter (T, F, P, N) respectively stands for $(true, false, positive, negative)$. *ncs* on TaskSanitizer stands for "no compiler support" and indicates that the test does not compile with Clang 8.x. *segv* on ROMP indicates that the instrumented execution was incomplete due to a run-time error.

As Taskgrind's main objective is to avoid false negative reports, we have highlighted them. Amongst all the tools, it reports the least false-negative with only a single one on *DRB129-mergeable-taskwait-orig*. It evaluates the support for OpenMP `mergeable` clause, which semantic is not supported by Taskgrind, nor by TaskSanitizer, Archer, or ROMP. Single-thread execution of TMB reports 100% accuracy, while other tools do not. The multi-threaded execution of TMP reports a few false positives that require further investigation: Taskgrind detects conflicting sibling tasks on a memory location in their parent segment stack frame.

B. LULESH

Table II and Fig. 4 reports data race detection and overheads on execution time and memory usage on the LULESH proxy application on 12th Gen Intel(R) Core(TM) i5-12450H. The execution time reported for Taskgrind only includes the recording phase and not the determinacy race analysis.

On the table, we report two versions of LULESH: a correct one (with *no* in the "racy" column) and an incorrect one (with *yes* in the "racy" column) by removing a task dependence to introduce data races intentionally. We also run it on 1 and 4 threads for the same reason as the previous experiment. On the single-threaded run, we observe a slowdown against non-instrumented versions of $10x$ for Archer and $100x$ for Taskgrind and a memory overhead of $4x$ for Archer and $6x$ for Taskgrind. The reasons for Taskgrind deadlocks running with multiple threads remain to be investigated. Regarding data race, Archer never reports errors when running in a single-thread, most likely because it only sees undeferred tasks through OMPT due to LLVM serialization. On the other hand,

[4]https://github.com/llvm/llvm-project/issues/89398

Micro-benchmarks suite		Tool			
DRB (with OMP_NUM_THREADS=4)	Determinacy Race	TaskSanitizer	Archer	ROMP	Taskgrind
027-taskdependmissing-orig	yes	TP	**FN**	TP	TP
072-taskdep1-orig	no	TN	TN	TN	TN
078-taskdep2-orig	no	TN	TN	TN	FP
079-taskdep3-orig	no	*ncs*	TN	TN	FP
095-doall2-taskloop-orig	yes	*ncs*	TP	TP	TP
096-doall2-taskloop-collapse-orig	no	*ncs*	TN	TN	FP
100-task-reference-orig	no	*ncs*	FP	TN	FP
101-task-value-orig	no	FP	FP	TN	FP
106-taskwaitmissing-orig	yes	TP	TP	TP	TP
107-taskgroup-orig	no	FP	TN	TN	FP
122-taskundeferred-orig	no	FP	TN	FP	TN
123-taskundeferred-orig	yes	TP	TP	TP	TP
127-tasking-threadprivate1-orig	no	*ncs*	TN	*segv*	FP
128-tasking-threadprivate2-orig	no	*ncs*	TN	TN	FP
129-mergeable-taskwait-orig	yes	*ncs*	**FN**	**FN**	**FN**
130-mergeable-taskwait-orig	no	*ncs*	TN	TN	TN
131-taskdep4-orig-omp45	yes	*ncs*	TP	TP	TP
132-taskdep4-orig-omp45	no	*ncs*	TN	TN	TN
133-taskdep5-orig-omp45	no	*ncs*	TN	TN	TN
134-taskdep5-orig-omp45	yes	*ncs*	TP	TP	TP
135-taskdep-mutexinoutset-orig	no	*ncs*	TN	FP	TN
136-taskdep-mutexinoutset-orig	yes	TP	TP	TP	TP
165-taskdep4-orig-omp50	yes	*ncs*	**FN**	TP	TP
166-taskdep4-orig-omp50	no	*ncs*	TN	TN	TN
167-taskdep4-orig-omp50	no	*ncs*	TN	TN	TN
168-taskdep5-orig-omp50	yes	*ncs*	TP	TP	TP
173-non-sibling-taskdep	yes	**FN**	**FN**	**FN**	TP
174-non-sibling-taskdep	no	TP	TN	TN	FP
175-non-sibling-taskdep2	yes	**FN**	TP	TP	TP
TMB (OMP_NUM_THREADS=1)					
1000-memory-recycling.1	no	TN	TN	TN	TN
1001-stack.1	yes	TP	**FN**	**FN**	TP
1002-stack.2	no	TN	TN	TN	TN
1003-stack.3	no	FP	TN	TN	TN
1004-stack.4	yes	TP	**FN**	TP	TP
1005-stack.5	no	FP	TN	TN	TN
1006-tls.1	no	FP	TN	TN	TN
TMB (OMP_NUM_THREADS=4)					
1000-memory-recycling.1	no	TN	TN	TN	FP
1001-stack.1	yes	TP	**FN/TP**	TP	TP
1002-stack.2	no	TN	TN	TN	FP
1003-stack.3	no	TN	TN	TN	TN
1004-stack.4	yes	TP	TP	TP	TP
1005-stack.5	no	TN	TN	TN	TN
1006-tls.1	no	FP	TN	TN	FP

TABLE I: Micro-benchmark suites results

		Execution time (s.)			Memory usage (MB.)			N° of reports	
Racy	N° of threads	No tools	Archer	Taskgrind	No tools	Archer	Taskgrind	Archer	Taskgrind
no	1	0.01	0.12	1.23	10	41	64	0	0
	4	0.01	0.43	*deadlock*	15	83	*deadlock*	149 to 273	*deadlock*
yes	1	0.01	0.12	1.23	10	41	64	0	458
	4	0.01	0.46	*deadlock*	15	84	*deadlock*	140 to 221	*deadlock*

TABLE II: Execution time, memory usage overheads and number of reports for Archer and Taskgrind, on a dependent task-based OpenMP implementation of LULESH with -s 16 -tel 4 -tnl 4 -p -i 4

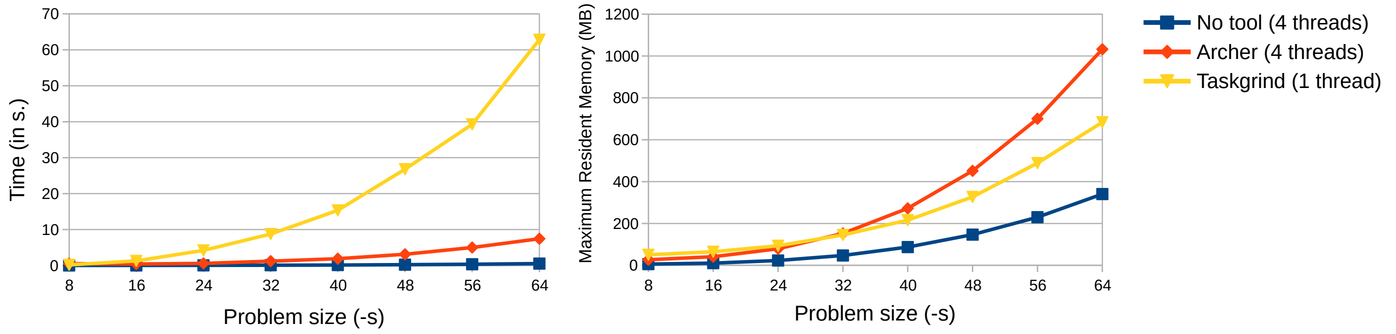


Fig. 4: Execution Time and Memory overheads on LULESH with parameters '-s $s -tel 4 -tnl 4 -p -i 4'

we annotated the code to notify Taskgrind that the task is semantically deferrable, which leads to no false negatives on the racy version with 458 races detected.

On the figure, the x-axis varies the problem size `-s` - with the application having a $O(s^3)$ time and memory complexity. The reference and Archer execute with four threads, and Taskgrind with a single thread.

We also evaluated using ROMP but omitted results from the figure, as the instrumented program crashed early during the first iteration of LULESH. Still, measurements showed that memory use and execution time follow a similar shape with `s`, but with much larger overheads: for `-s=64`, before the application crashes, the execution time and memory use were respectively $79s$ and $75GB$.

C. Error Reporting

Listing 4 is a minimal erroneous OpenMP program, where the tasks line 8 and 11 may both concurrently access x. Listing 5 and Listing 6 respectively transcribe errors reported by ROMP and Taskgrind, compiling with debug symbols (option `-g`). As you can see, Taskgrind provides debug information thanks to the Valgrind framework, while ROMP does not by default even though its DBI framework (Dyninst) can be patched to enable similar report.

```
1  int main(void)
2  {
3      int * x = (int *) malloc(2 * sizeof(int));
4      # pragma omp parallel
5      {
6          # pragma omp single
7          {
8              # pragma omp task
9                  x[0] = 42;
10
11             # pragma omp task
12                 x[0] = 43;
13         }
14     }
15     return 0;
16 }
```

Listing 4: task.c - an errorneous OpenMP Program

```
1  [...]
2  data race found:
```

Listing 5: An error report from ROMP

```
1  Segments task.1.c:8 and task.1.c:11 were declared
        independent while accessing the same memory
       address
2  4 bytes from 0xC3EA040 allocated in block 0
       xC3EA040 of size 8
3  from task.1.c:3
4  [...]
```

Listing 6: An error report from Taskgrind

VI. Related Work

Taskgrind relies on Valgrind to instrument memory accesses and attach them to segments, it eventually implements a determinacy race analysis based on these information. We present a brief review of the literature on programs instrumentation framework and determinacy races analysis.

a) Instrumentation Frameworks: Instrumenting programmes to analyse them is a much-studied subject in the literature, and several approaches had been explored. Pin [16] and Valgrind [9] rely on Dynamic Binary Translation (DBT) [17]: the binary program is translated to an intermediate representation, instrumented, and its execution is emulated keeping track of a virtual machine state. The main drawback is the cost of translating and emulating, that lead to important slowdown when executing. It motivates the design of instrumentation frameworks such as DynamoRIO [18], Dyninst [8] or Instrew [19], which is capable of instrumenting while executing natively. All these frameworks present different capabilities, in the end, we choose Valgrind for its well-established capabilities (memory allocator overloading, memory access instrumentation, debug information reading...) and for its active community.

b) Determinacy Race Analyzer: In 1997, Nondeterminator [20] was designed as an entire toolsuite (including compile-time instrumentation and run-time analysis) to detect accurately and provably determinacy races of Cilk programs. The core of its analyzer relies on a low complexity algorithm (SP-Bags) which assumes the program serial execution correct - known as the *serial elision*. Taskgrind has no such assumption.

Helgrind+ (2007-2009) is a Valgrind tool for data race detection. It supports pthread synchronizations primitives, including mutexes and condition variables, which are currently not supported by Taskgrind but could be added in the future.

ThreadSanitizer [21] (2009) was originally designed as a Valgrind tool and support pthread-based synchronizations. Its core moved to the LLVM compiler [22] to reduce execution overheads by instrumenting during the compilation and executing natively.

R. Raghavan et al. [23] (2010) generalized Nondeterminator approach for async/await programming models, and provided a tool for X10 programs, still under the serial elision assumption.

Archer [3] was introduced in 2016 as a ThreadSanitizer extension to support OpenMP semantics. However, it is a thread-centric analyzer that may be a source of false negatives running under task-related synchronizations which is a source of motivations for TaskSanitizer [4], ROMP [7] and Taskgrind.

OmpSs-2 has a toolchain to verify the parallelization of programs [24]. It checks five errors (E1, E2, E3, E4, E5) "*using local task analysis to deal with each task separately*" - as opposed to Taskgrind which looks at the program entirely through its segment graph. If OmpSs-2 were to be supported by Taskgrind, errors E3, E4 and E5 would fallback to a generic determinacy-error. On the other hand, OmpSs-2 toolchain seems to provide more insights about the error, such as synchronizations mechanism suggestions. Extending Taskgrind to report programming model-specific suggestion is left as future work, and could assist programmers in parallelizing their applications.

VII. Conclusion and Future Work

This paper introduced Taskgrind[5]: a Valgrind tool for parallel program memory access analysis. It analyses segment graphs that model parallel programs. We implemented a determinacy race analysis, as it sometimes hints at correctness errors of asynchronous task-based programs while also addressing some pitfalls bound to heavyweight dynamic binary instrumentation (DBI). As opposed to source code compile-time instrumentation (such as LLVM Sanitizers), heavyweight DBI ensures the instrumentation of every program instruction (even closed-source) - aiming at no false-negative report due to non-instrumented segments. However, it required a lot of engineering to filter out the many false positives that arose and provide meaningful report to programmers. In this paper, we have illustrated a few, such as the inherent non-determinacy of parallel runtime systems, memory allocator recycling, and thread/segment-local accesses.

In the future, we would like to implement support for Cilk, Qthreads, and explore the support of GPU programming models. In addition, a few drawbacks of the current design could be improved:

- On memory recycling, (1) removing free operations is not a satisfactory solution, and (2) we need to support libraries built-in memory allocators.
- The determinacy race post-processing analysis is an embarrassingly parallel algorithm, but it is currently run sequentially within the Valgrind framework after the instrumented program execution.

[5]https://gitlab.inria.fr/ropereir/valgrind

Finally, we would like to add more analysis and improve error reporting, having Taskgrind move toward a more general "trial and error" parallel programming assistant.

References

[1] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in cilk programs," in *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 1–11. [Online]. Available: https://doi.org/10.1145/258492.258493

[2] U. Bora, S. Das, P. Kukreja, S. Joshi, R. Upadrasta, and S. Rajopadhye, "Llov: A fast static data-race checker for openmp programs," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, dec 2020. [Online]. Available: https://doi.org/10.1145/3418597

[3] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "Archer: Effectively spotting data races in large openmp applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 53–62.

[4] H. S. Matar and D. Unat, "Runtime determinacy race detection for openmp tasks," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 31–45.

[5] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with llvm compiler," in *Runtime Verification*, S. Khurshid and K. Sen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 110–114.

[6] E. W. Dijkstra, "On the reliability of programs," n.d., circulated privately. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF

[7] Y. Gu and J. Mellor-Crummey, "Dynamic data race detection for openmp programs," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 767–778.

[8] W. R. Williams, X. Meng, B. Welton, and B. P. Miller, "Dyninst and mrnet: Foundational infrastructure for parallel tools," in *Tools for High Performance Computing 2015*, A. Knüpfer, T. Hilbrich, C. Niethammer, J. Gracia, W. E. Nagel, and M. M. Resch, Eds. Cham: Springer International Publishing, 2016, pp. 1–16.

[9] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, p. 89–100, jun 2007. [Online]. Available: https://doi.org/10.1145/1273442.1250746

[10] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt: An openmp tools application programming interface for performance analysis," in *OpenMP in the Era of Low Power Devices and Accelerators*, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 171–185.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 212–223.

[12] T. B. Schardl and I.-T. A. Lee, "Opencilk: A modular and extensible software infrastructure for fast task-parallel code," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 189–203.

[13] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[14] U. Drepper, "Technical Report - ELF Handling For Thread-Local Storage," 2013.

[15] P.-H. Lin and C. Liao, "High-precision evaluation of both static and dynamic tools using dataracebench," in *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, 2021, pp. 1–8.

[16] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:6719639

[17] M. Probst, "Dynamic binary translation," 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:16335388

[18] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 133–144. [Online]. Available: https://doi.org/10.1145/2151024.2151043

[19] A. Engelke, D. Okwieka, and M. Schulz, "Efficient llvm-based dynamic binary translation," in *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 165–171. [Online]. Available: https://doi.org/10.1145/3453933.3454022

[20] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in cilk programs," 1997.

[21] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 62–71. [Online]. Available: https://doi.org/10.1145/1791194.1791203

[22] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with llvm compiler," in *Runtime Verification*, S. Khurshid and K. Sen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 110–114.

[23] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Efficient data race detection for async-finish parallelism," in *Runtime Verification*, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 368–383.

[24] S. Economo, S. Royuela, E. Ayguadé, and V. Beltran, "A toolchain to verify the parallelization of ompss-2 applications," in *Euro-Par 2020: Parallel Processing*, M. Malawski and K. Rzadca, Eds. Cham: Springer International Publishing, 2020, pp. 18–33.

APPENDIX

List of software versions used in this paper.

- GNU/Linux 6.5.0-44-generic x86_64
- LLVM/Archer releases 8.x, 14.x, 17.x and 19.x
- TaskSanitizer (commit a3d3b44)
- ROMP (commit 4ea4ad5b)
- Taskgrind SC-24 (https://gitlab.inria.fr/ropereir/valgrind)