# ORAQL — Optimistic Responses to Alias Queries in LLVM

1st Jan Hückelheim
*Argonne National Laboratory*
Lemont, IL, USA
jhueckelheim@anl.gov

2nd Johannes Doerfert
*Argonne National Laboratory*
Lemont, IL, USA
jdoerfert@anl.gov

***Abstract*—Alias information is a prerequisite to many compiler optimizations. Since alias analysis is generally undecidable, implementations rely on conservative heuristics that are limited by compile time constraints. Pointers conservatively reported as may-alias may prevent subsequent optimizations, and alias analyses are thus an active research area.**

**With ORAQL, we present an optimistic (rather than conservative) alias analysis in LLVM. While this does not result in a sound compilation, it allows determining potential gains of better alias analyses in order to guide the use of program annotations and modifications and compiler research.**

**ORAQL is implemented as a a driver, test script, and analysis pass that collaborate during compilation to find a locally maximal set of queries that can be answered "no-alias" without breaking tests. Our results show that HPC proxy applications compiled with LLVM, regardless of the programming language and parallel programming model, are often not limited by alias information.**

***Index Terms*—alias analysis, performance gap estimation, compiler optimization**

## I. Introduction

Alias analysis (AA) is an often cited reasons for performance, or the lack thereof. However, it is also an easy scapegoat rather than the culprit identified by a proper analysis. Pointing at alias analysis allows to justify differences in compiler effectiveness, the runtime gap between "low"- and "high"-level languages, missing transformations, and a myriad of otherwise hard to explain behaviors. Colloquial wisdom is that analysis analysis is hard, in-fact it is undecidable [1], and at the same time crucial for performance, at least that is what is often believed. It is consequently not surprising that compilers and programming languages adopted various ways to reason about aliasing and the absence of it. Examples include:

- the strict aliasing rules in C/C++,
- the [__]restrict annotations in C/C++,
- the pointer and target attributes in Fortran,
- the -f[no-]alias command line option in Intel's ICC, and
- seven distinct alias analyses in LLVM 14 ({Basic, ScopedNoAlias, TypeBased, ObjCARC, Globals, CFLAnders, CFLSteens}AA).

In this work we develop a generic methodology to analyze and attribute the effect of alias analyses queries, and we provide a way to determine an estimate of the achievable performance with (almost) optimal alias information. In addition to the overall efficiency gap, we can automatically pinpoint the alias queries responsible for the most severe performance degradation in a program. Further, our approach allows to measure how different programming languages and parallelization extensions, are impacted by alias information.

In contrast to classical alias analyses that try to proof the absence of aliasing (or dependences) through static, dynamic, or hybrid reasoning, our work is not determining a sound result for all possible program inputs. Instead, we ensure that the output for a set of user provided inputs does not change. Hence, our *optimistic alias query responses* are either correct or effectively irrelevant, for the given set of user inputs, and we cannot reason about inputs not in the set. As such, *ORAQL* is not a static program analysis to be used in a production run but rather a development tool to determine the impact of imperfect alias information, as well as the locations and reasons where potential aliasing is causing the most performance degradation.

There are various use cases for ORAQL, including:

1) Sporadic source code tuning by developers that are willing to modify and annotate their program to achieve better performance. In contrast to unguided annotation, e.g., marking all arguments as restrict, the ORAQL workflow allows to minimize changes to the program while retaining most benefits. Given that any annotation induces maintenance cost, e.g., the invariant has to be preserved over time or the annotation needs to be removed, it is crucial for productivity to avoid an overload of annotations.
2) Compiler development can benefit through ORAQL as it identifies the most important missed cases. By focusing on the most important kinds of conservatively answered aliasing queries one can provide specialized analyses and representations for information. The goal is to ensure their impact is known to be useful in practice, not only on contrived examples, while at the same time their overhead is kept lower than generic alternatives. As the most costly alias analysis are often disabled by default due to their scaling behavior, specialized versions that cover the most important cases provide a new trade-off not available so far.

3) With the ORAQL result, the search space for alias analysis-related tuning techniques has known bounds. Selecting the appropriate subset of analyses for a program or domain (e.g., out of the seven provided by LLVM 14), and the best values for their respective hyper-parameters was in practice done by hand. However, a bounded search space allows to perform faster tuning that stops if the improvement potential becomes negligible. In addition to new optimization pipelines this also allows to revisit the current compiler defaults and to make an informed decision based on potential command line options (such as O1, O2, O3, Ofast) and input programming language.

The rest of this paper is structured as follows: In Section II the contributions and limitations are clarified before some necessary background is discussed in Section III. The design of ORAQL is explained in Section IV. An evaluation of ORAQL on seven HPC proxy applications in different configurations is provided in Section V followed by a summary of the lessons learned in Section VI. The paper concludes with VIII after related work was discussed in Section VII.

## II. Contributions and Limitations

This work introduces ORAQL, a tool to automatically identify (almost) perfect alias information for a given program on a given set of inputs. The main contributions are:

- The ORAQL tool, consisting of a LLVM alias analysis pass, a driver, and a test script. The fully automatic tool takes a program, compilation instructions, and a test suite to produce (almost) perfect alias query responses.
- ORAQL further generates a program compiled with (almost) perfect alias information, and, if requested, a report identifying the optimistically and forced pessimistically answered alias queries. These queries are associated with source lines, where possible, and with the passes that issued them.
- An evaluation of ORAQL on seven HPC proxy applications. Four of these applications are evaluated in different configurations. Some configurations utilize OpenMP, Kokkos, or MPI, while others are re-implementations in different programming languages.

ORAQL is a research prototype with limitations, including:

- ORAQL is not a program analysis for every day use. The results are only meaningful for pre-determined inputs and the tested compiler (version). However, LLVM-based (vendor) compilers can be compatible if they choose to be.
- The user is required to provide a configuration file that could be automatically generated from common build, test, and profiling steps, e.g., using cmake.

## III. Background

The LLVM compiler contains a large number of passes that are called in sequence by the pass manager. Passes may be analysis or transformation passes. Both types of passes may utilize analysis information computed by previous passes, for example to improve their own precision or effectiveness. Likewise, passes may affect subsequent passes, for example by providing additional analysis information, invalidating previously available information during a code transformation, or changing the code structure in a way that enables or simplifies subsequent analyses.

Since passes must be conservative and run time and memory efficient, they often implement heuristics whose effectiveness depends in non-intuitive ways on the available information and thus on the order in which passes are executed. Therefore, a more precise (or more optimistic) alias analysis may lead to performance improvements, for example by enabling beneficial transformations, or may lead to performance degradation, for example by enabling a transformation that then prevents other, even more beneficial transformations from occurring. More information may in some cases not have a noticeable effect on overall performance, for example because the information does not actually enable additional impactful transformations, or enables transformations in code paths that are never or rarely taken. Similarly, optimistic information might be used by an analysis, e.g., memory SSA [2], but its result may or may not impact an actual transformation. Summarized, more (including optimistic) information in a compiler does not guarantee any change to the executable, nor do changes imply it is going to perform better.

One class of analysis is *alias analysis*. The goal is to determine whether two pointers *alias*, or in other words, whether they point to the same or overlapping memory addresses. Alias analysis typically results in one of three answers: *must-alias*, which means that two pointers are guaranteed to point to the same or overlapping memory, *no-alias*, meaning that they are guaranteed not to point to the same or overlapping memory, and *may-alias*, indicating that their alias status was not determined with certainty. Note that every pointer pair either does or does not alias, and a *may-alias* response is a *pessimistic* answer indicating the lack of precise knowledge. In contrast, we consider *no-alias* to be the best case as it has the greatest chance of enabling subsequent transformations. This is because not all existing LLVM alias analysis passes are able to determine must-alias, and consequently must-alias results are not exploited by many transformations. Answering *no-alias*, even in the absence of sound guarantees, is therefore likely the most *optimistic* option.

Alias analysis in LLVM is lazily called when alias information is required for a certain pointer pair. The pass manager calls one alias analysis pass at a time in a predetermined sequence. A result is returned as soon as a pass in the sequence responds with a definite no-alias or must-alias answer. Otherwise, may-alias will be returned. Thus, may-alias is the pessimistic fallback if all passes in the sequence have responded may-alias. By appending the ORAQL alias analysis pass (ref. Section IV-A) to the end of this sequence, we ensure that we only act on the alias queries that can not be successfully analyzed by the other passes.

## IV. Approach

ORAQL is implemented as a collaboration between three components: an alias analysis pass within LLVM, a probing

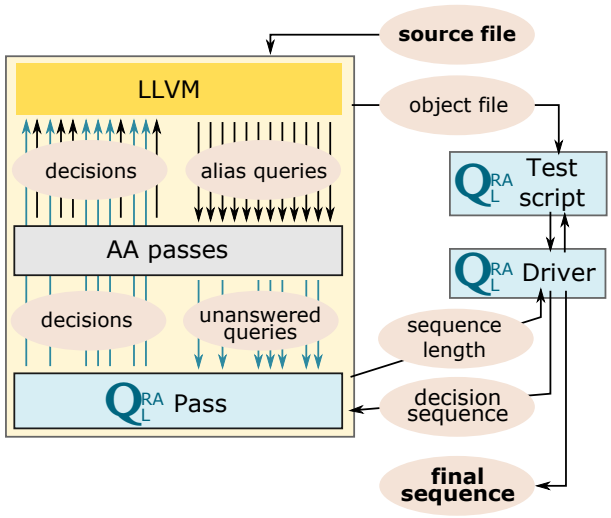


Fig. 1: Overall work flow of ORAQL. All parts on the left (yellow box) are within LLVM. Alias queries that can not be answered by existing analysis passes are given to the ORAQL pass. The pass answers these queries based on the decision sequence provided by the external ORAQL driver. The driver repeatedly compiles a given file with different decision sequences, until a maximally optimistic sequence is found that still allows the compiled executable to satisfy the user-defined criteria during tests by the ORAQL test script.

driver, and verification script, the latter two implemented in Python.

### A. ORAQL Alias Analysis Pass

"Alias analysis pass" is actually a misnomer, since no analysis is performed. Rather, the pass answers queries solely according to a predetermined response sequence. The ORAQL pass is appended as the final alias analysis pass in the LLVM pass manager, meaning that ORAQL only responds to those queries that can not be conclusively answered by any of the previously existing alias analysis passes.

Each alias query in LLVM consists of a pointer pair – representing the two memory addresses whose alias status is in question – as well as a location description for each pointer, which controls whether the query is just about the exact memory address or a larger range. To reduce the sequence length that needs to be probed, as well as to avoid inconsistencies in our optimistic responses which often violate LLVM's internal assumptions, the ORAQL pass stores a cache of previously seen queries and the given response. Any query identical to one previously seen will be served from the cache. Queries are considered identical if they pertain to the same pointer pair, regardless of the associated location descriptions. This allows us to further reduce our search space, but means that we are not able to identify cases where optimistic responses would be possible only for a certain location description of a pointer pair. In addition, ORAQL will not try to identify global maximal sets of queries and instead fix all optimistic choices after a successful verification. This could theoretically cause us to drop beneficial sequences as queries are not independent but can be arbitrarily connected. However, given the large number of optimistic choices that we make in our test cases, we believe that probing additional information or sequence orders is unlikely to yield substantial benefits.

The ORAQL pass is controlled by the probing driver through command line arguments. The most important input is the probing sequence, which is communicated as a series of space-separated "1" (optimistic, no-alias) and "0" (not optimistic, may-alias) characters. This sequence is passed through the argument `-opt-aa-seq=<sequence>`. Because sequences may be longer than the command line argument length limit, the probing script uses a functionality in the LLVM arguments parser that allows passing arguments through a file via `@<filename>`. Whenever the ORAQL pass is queried, it will first attempt to answer from cache, and otherwise consume a number from the sequence and respond according to that number. If the end of the sequence is reached, all subsequent unique queries are answered optimistically. The statistics reporting functionality in LLVM is used to communicate the number of unique (non-cached) queries to the driver, which will be used by the driver to adjust the sequence length if necessary.

### B. ORAQL Probing Driver

The probing driver relies on a benchmark-specific file that determines the compiler frontend (clang, clang++, various MPI wrappers, etc), compiler flags, and the exact files or functions to which optimistic probing is applied. The file also specifies a reference output file that is used by the verification script.

The probing driver starts by compiling and running the program with deactivated ORAQL pass, and calls the verification script to ensure a correct compilation and runtime behavior. After this, the driver attempts to answer all queries optimistically by activating the ORAQL pass with an empty sequence (which causes all queries to be answered with no-alias). If this succeeds, the driver reports this fact and returns.

If the verification script reports an error, the driver will determine the current sequence length based on the reported number of unique queries from the pass, and recursively bisect in order to identify the queries that have to be answered conservatively for the tests to pass. In our preliminary experiments, we found that most queries can be answered optimistically without breaking the compiled programs. In such a case, a recursive strategy is superior to one where each query is tested individually. Figure 2 illustrates recursive probing for a small example, and Figure 1 gives on overview of the collaboration between ORAQL components, as well as their interaction with other LLVM passes.

To reduce the number of tests that need to be executed, the driver stores hashes of all previously seen executables, along with their test results. If a sequence leads to an executable that is bit-identical to a previously seen one, the test is skipped and the previous result is returned from cache.

We implemented two bisection strategies. The first is implemented in "frequency space": Instead of splitting a sequence

of queries into two consecutive halves, the sequence is split according to the integer division remainder (for example, even/odd numbers in the first bisection level). This allows the driver to generate sequence descriptors that are independent of the sequence length and thus can be used without first determining the number of queries that can be answered, which simplifies the implementation.

However, preliminary experiments showed that "important" queries – those that break the tests if answered optimistically – are often clustered together. In these cases the frequency space probing must bisect to a level at which almost every query is tested individually, resulting in a large overall probing time. To resolve this, we implemented a "chunked" probing strategy. Intuitively, if a given sequence can not be answered optimistically, the strategy recursively calls itself twice; first on the part of the sequence corresponding to earlier queries, then on the part of the sequence corresponding to later queries. Since the overall number of queries can change depending on the preceding responses, the bisection strategy must adapt accordingly.

We note that our driver attempts to maximize the number of optimistic queries without measuring or modeling the performance impact of each query. Furthermore, any greedy search strategy, such as the ones implemented in our driver, is not guaranteed to find a global optimum. For example, replying optimistically to one query may preclude an optimistic answer to another, more beneficial query if answering both of them optimistically will break the program. For this reason, we refer to the final sequence discovered by the probing script as *almost* optimal. Global optimization would require not only checking each query individually, but instead checking every combination of alias responses.

### C. ORAQL Verification Script

Before running ORAQL, the user has to obtain a reference output. We did this by compiling each benchmark with LLVM without ORAQL. All our benchmarks print relevant figures to the standard output and in most cases self-diagnose with a checksum of their final result, allowing us to use the `stdout` of our benchmarks for verification purposes. However, we observe that the output of our benchmarks varies slightly between runs. For example, most benchmarks report a run time (which is noisy and changes each time), and the checksums and printed numbers may differ slightly between test case variants and on different machines. For example, the LULESH test case creates a different-size mesh depending on the number of MPI ranks and whether or not OpenMP is used, and some test cases show slight numerical variations in the least significant digits of the printout between runs even when ORAQL is not used. We therefore created multiple reference outputs for some test cases, and use regular expressions where appropriate to ignore certain parts of the output during verification. It is worth noting that our tests only use concrete inputs and do not necessarily cover all parts of the executable. Therefore, some optimistic alias responses will affect code paths that are not executed and will thus not affect performance.

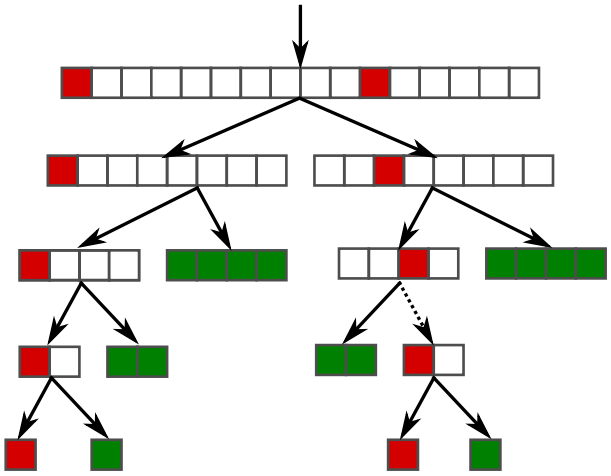

Fig. 2: Probing in ORAQL driver. The top row represents an entire sequence of alias queries that are unanswered by the other analysis passes and thus passed to ORAQL. Most of the queries, represented by white squares, can be answered optimistically without leading to failed tests, while few queries, represented by red squares, can not. ORAQL will recursively refine the responses to pinpoint those "dangerous" queries. The dotted arrow represents a test whose outcome can be deduced from the fact that the parent contained a dangerous query, while the sibling did not. Using this fact can reduce the number of tests that need to run.

### D. Static Impact Identification

Many programs can be successfully verified with significant optimistic knowledge. As such, it is often more informative to identify and inspect the "true aliases" that caused ORAQL to respond to a query pessimistically. The ORAQL compiler pass offers four flags to dump information about the decisions in a way that allows users to associate them with the actual program source. The command line flags `-opt-aa-dump-{first,cached}` determine if initial or cached queries should be printed, while the flags `-opt-aa-dump-{optimistic,pessimistic}` determine if optimistically or pessimistically answered queries should be printed. At least one of each category is necessary to obtain an output. In addition to information about the queries it is often interesting to put them into context. We leverage existing LLVM functionality to associate the ORAQL output with the analysis or transformation pass that requested it. As an example, Figure 3 shows all four pessimistically answered non-cached queries for the TestSNAP OpenMP version (ref. Section V-A), as well as the pass that queried them. The latter is determined as the last pass that was executed before the queries were printed. LLVM allows to print pass names prior to their execution with the `-debug -pass=Executions` flag. Note that due to the caching of queries in the pass the initial queries might not be the ones that caused a verification failure if answered optimistically in isolation. Put another way, the pass that queried a pessimistic

```
[...] Executing Pass 'Global Value Numbering' on Function
↪  '.omp_outlined._debug__.6'...
[ORAQL] Pessimistic query [Cached 0]
[ORAQL] - %9 = load double*, double** %dptr.i96, align 8, !tbaa !1891
↪  [LocationSize::beforeOrAfterPointer]
[ORAQL] - %class.SNA* %this [LocationSize::beforeOrAfterPointer]
[ORAQL] Scope: .omp_outlined._debug__.6

[ORAQL] Pessimistic query [Cached 0]
[ORAQL] - %class.SNA* %this [LocationSize::beforeOrAfterPointer]
[ORAQL] - %15 = load i32*, i32** %dptr.i141, align 8, !tbaa !1916
↪  [LocationSize::beforeOrAfterPointer]
[ORAQL] Scope: .omp_outlined._debug__.6

[ORAQL] Pessimistic query [Cached 0]
[ORAQL] - %50 = load %struct.SNAcomplex*, %struct.SNAcomplex** %dptr.i135, align 8,
↪  !tbaa !1941 [LocationSize::beforeOrAfterPointer]
[ORAQL] - %52 = load %struct.SNAcomplex*, %struct.SNAcomplex** %dptr.i117, align 8,
↪  !tbaa !1943 [LocationSize::beforeOrAfterPointer]
[ORAQL] Scope: .omp_outlined._debug__.6

[ORAQL] Pessimistic query [Cached 0]
[ORAQL] - %re90 = getelementptr inbounds %struct.SNAcomplex, %struct.SNAcomplex* %50,
↪  i64 %idxprom.i82, i32 0, !dbg !2051 [LocationSize::precise(8)]
[ORAQL] - %re104.1 = getelementptr inbounds %struct.SNAcomplex, %struct.SNAcomplex*
↪  %52, i64 %idxprom.i71.1, i32 0, !dbg !2063 [LocationSize::precise(8)]
[ORAQL] Scope: .omp_outlined._debug__.6
[ORAQL] LocA: sna.cpp:609:60
[ORAQL] LocB: sna.cpp:614:46
```

Fig. 3: ORAQL debug output during the compilation of sna.cpp, the main source file of TestSNAP (ref. Section V-A), when compiled with OpenMP enabled. The snippet shows all four pessimistically answered queries but not repeated responses when they were served from the pass's query cache (ref. Section IV-A). The first line shows the pass that issued the queries, here Global Value Numbering (GVN). A query always shows the response kind, the cache status, the two pointers and the location description. Other information about the query is provided if available, including the name of the function that contained (at least one of) the pointers. When debug information is present the source locations of the two pointers is shown, e.g., in the last query the pointers originate from sna.cpp line 609 and 614, respectively. The function in question is not in the source code but generated by the OpenMP frontend, thus, it is initially a parallel region. Given the line numbers we can manually determine that the parallel region itself is contained in `SNA::compute_deidrj`.

response first might not be the one that acted on the otherwise optimistic information in a way that affected the program output. Any cached version of the query might have resulted in the change that caused the verification failure.

### *E. Multi Target Compilation*

The prevailing compilation model for accelerators nowadays uses a single source file that contains code for the host and the devices. That usually means compilers effectively run multiple times on a source file, once per target architecture. To simplify the use of ORAQL for offloading use cases, e.g., CUDA and OpenMP offload, we introduced a command line flag that will restrict ORAQL with regards to the currently targeted architecture. This shorthand, `-opt-aa-target=<target-sub-string>`, makes it easy to use ORAQL only for the accelerator part of a compilation. That said, if ORAQL is used for both parts we currently cannot adjust the sequence between the different compilations run of the same file. Effectively, ORAQL will reuse the sequence for all targets which will lead to a pessimistic intersection of all of them.

## V. Evaluation

We evaluated ORAQL across 7 HPC proxy applications, most of them in various configurations. Overall, we utilized four different parallel programming models as well as four programming languages in our experiments. The experiments included host and offload runs. The former were performed on Intel(R) Xeon(R) Platinum 8180M CPU @ 2.50GHz (Skylake) CPUs with 112 threads 768GB memory. The latter were executed on a NVIDIA A100 GPU. Depending on the experiment we report number of executed instructions, wall clock time,

| Benchmark | Programming Model | Source Files | # Opt. Queries | | # Pess. Queries | | # No-Alias Results | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Unique | Cached | Unique | Cached | Original | ORAQL | Δ |
| TestSNAP | C++ | sna | 30101 | 38076 | 0 | 0 | 44259 | 95487 | +115.7% |
| TestSNAP | C++, OpenMP | sna | 3856 | 12514 | 4 | 265 | 19152 | 34425 | +79.7% |
| TestSNAP | C++, Kokkos, CUDA | sna | 9110 | 54192 | 0 | 0 | 118623 | 149525 | +26% |
| TestSNAP | Fortran | all (manual LTO) | 32810 | 52539 | 237 | 69 | 377862 | 478249 | +26.5% |
| XSBench | C | Simulation | 415 | 168 | 11 | 1 | 9954 | 10522 | +5.7% |
| XSBench | C, OpenMP | Simulation | 546 | 1294 | 11 | 1 | 12131 | 13480 | +11.1% |
| XSBench | CUDA, Thrust | Simulation | 3731 | 16734 | 11 | 1 | 33312 | 53942 | +43.1% |
| GridMini | C++, OpenMP Offload | Benchmark_su3 | 86 | 6809 | 0 | 0 | 8969 | 14435 | +60.9% |
| Quicksilver | C++, OpenMP | all (manual LTO) | 31312 | 68542 | 0 | 0 | 135504 | 242001 | +78.5% |
| LULESH | C++ | lulesh | 30810 | 188826 | 35 | 131 | 416371 | 668864 | +60.64% |
| LULESH | C++, OpenMP | lulesh | 29981 | 128537 | 15 | 0 | 195724 | 385730 | +97.1% |
| LULESH | C++, MPI | lulesh | 28832 | 160032 | 99 | 207 | 356965 | 555141 | +55.5% |
| MiniFE | C++, OpenMP | main | 6592 | 10852 | 58 | 142 | 134567 | 149912 | +11.4% |
| MiniGMG | C, OpenMP | operators.ompif | 36080 | 23235 | 0 | 0 | 124431 | 198012 | +59.1% |
| MiniGMG | C, OpenMP tasks | operators.omptask | 33007 | 21845 | 0 | 0 | 121110 | 186836 | +54.2% |
| MiniGMG | C, SSE intrinsics | operators.sse | 36166 | 32529 | 0 | 0 | 116700 | 200120 | +71.5% |

Fig. 4: Alias query statistics for all benchmarks and configurations (programming languages and parallel programming models). The *# Opt. Queries* and *# Pess. Queries* columns show the number of queries answered by the ORAQL pass either optimistically or pessimistically using the final sequence. The number of optimistically answered queries allows to put the pessimistic one, if any, into context. The rightmost columns show the overall number of no-alias responses across all alias analysis passes (not just ORAQL). Similarly, the original count of no-alias results allows to judge the overall percentage of optimistic queries required to get (almost) perfect alias information.

figure of merit results, or GPU kernel times. The software version we used are shown in Figure 5. The benchmarks are briefly summarized in Figure 4. Given that we modified the benchmarks slightly, e.g., their build system and printed output, to ease our evaluation we will bundle them with the version of ORAQL we used in a publicly available repository.

In the following we will introduce the benchmarks briefly and summarize our findings for each of them. Depending on the ORAQL results we performed different follow up investigations rather than a simple performance comparison across the board. One of our goals is to show how different insights can be derived via ORAQL as the actual results will inevitably depend on the user code and compiler being used. Thus, the data is a snapshot in time rather than a fundamental property of the used benchmarks, programming models, languages, and compiler. While we focus on the raw data first, Section VI will summarize the results holistically.

| | |
|---|---|
| LLVM | **git** `ea7be7e` |
| LLVM/Flang (fir-dev) | **git** `972e1f8` |
| Legacy Flang | **git** `b90b722` |
| CUDA | `11.4.0` |
| Kokkos | `3.5.0` |

Fig. 5: Software versions used as basis of ORAQL as well as to compile and run the various benchmarks. The benchmarked code, including minor modifications and ORAQL related scripts, is available in here

### A. TestSNAP

TestSNAP is a proxy for the SNAP force calculation in the LAMMPS molecular dynamics package [3–6]. It contains synthetic inputs (neighbor atom positions) and reference outputs (forces on atoms) for several different SNAP models. Force calculation is performed repeatedly before grind time (msec/atomstep) and RMS force error (eV/A) are reported.

We used four versions of TestSNAP for our experiments to show not only the applicability of ORAQL but also determine how much the programming language or parallel programming model impacts the result. The first three versions are C++-based, a sequential one, an OpenMP parallelized one, and a Kokkos version executed through the CUDA backend on a NVIDIA A100 GPU. For all we run ORAQL only on the sna.cpp file that contains the main computation kernel. The fourth version is a Fortran one compiled with the LLVM/Flang, or, to be more precise, the "fir-dev" branch. We used the math library distributed with the legacy Flang compiler (https://github.com/flang-compiler/flang.git). All source files have been compiled into LLVM-IR without explicit optimizations. We then linked them in a manual step into a single LLVM-IR bitcode file which we optimized with ORAQL as part of the regular `-O3` optimization pipeline.

*a) TestSNAP - Sequential:* The sequential version of TestSNAP was compiled fully optimistic without invalidating our verification constraints. In total, 68177 queries were answered as optimistic no-alias and 44% of these were unique. In the original version only 44259 alias queries were answered with no-alias, thus ORAQL raised that number by 115%. Other notable differences in the statistics reported by LLVM are shown in Figure 6. The number of executed instructions,

| Benchmark | Pass | Property | LLVM Statistics Output | | |
|---|---|---|---|---|---|
| | | | Original | ORAQL | Δ |
| XSBench - C++ | asm printer | # machine instructions generated | 1763 | 1688 | -4.2% |
| XSBench - CUDA | early CSE | # instructions eliminated | 1482 | 1538 | +3.8% |
| TestSNAP - Kokkos | asm printer | # machine instructions generated | 8573 | 8309 | -3% |
| TestSNAP - Fortran | asm printer | # machine instructions generated | 57020 | 53487 | -6.1% |
| TestSNAP - Kokkos | LICM | # loads hoisted or sunk | 728 | 931 | +27.8% |
| TestSNAP - Fortran | LICM | # loads hoisted or sunk | 70 | 961 | +1272% |
| GridMini | LICM | # loads hoisted or sunk | 4 | 10 | +150% |
| Quicksilver | loop deletion | # deleted loops | 2 | 55 | +2650% |
| Quicksilver | DSE | # stores deleted | 6 | 98 | +1533.3% |
| Quicksilver | GVN | # loads deleted | 45 | 245 | +444.4% |
| Quicksilver | LICM | # loads hoisted or sunk | 5 | 21 | +320% |
| Quicksilver | register allocation | # register spills inserted | 780 | 757 | -2.9% |
| MiniFE | SLP | # vector instructions generated | 139 | 185 | +33% |
| MiniGMG - OpenMP | loop vectorizer | # vectorized loops | 9 | 12 | +33% |
| MiniGMG - OpenMP tasks | loop vectorizer | # vectorized loops | 9 | 11 | +22% |
| MiniGMG - SSE intrinsics | loop vectorizer | # vectorized loops | 11 | 13 | +18% |
| MiniGMG - OpenMP tasks | LICM | # loads hoisted or sunk | 208 | 366 | +75.9% |
| MiniGMG - OpenMP | LICM | # loads hoisted or sunk | 215 | 394 | +83.2% |
| MiniGMG - SSE intrinsics | LICM | # loads hoisted or sunk | 202 | 368 | +82% |

Fig. 6: Interesting statistics reported by LLVM via `-mllvm -stats` for the original and ORAQL compilation of the respective benchmark. A selection is shown because of the large number of statistics and the fact that many are hard to interpret as secondary effects impact them.

as reported by `perf`, was down by 1.2%. The performance improved even by 3.6%, both with the `-DREFDATA_TWOJ =14100` build parameter and `-ns 10` command line option.

*b) TestSNAP - OpenMP:* The OpenMP version of TestSNAP required four queries to be answered pessimistically for the verification to pass. All four originate from the parallel region inside `SNA::compute_deidrj` which was outlined by the OpenMP frontend. The first two compare the implicit `this` pointer argument of the C++ class `SNA` with the `dptr` (data pointer) variable of the employed array abstractions. The third query involves two `SNAcomplex` pointers loaded from different `dptr` instances, and the last one is concerned with two loop carried accesses to arrays of `SNAcomplex` values. While the Global Value Numbering (GVN) pass was the first to issue these four queries it might not have been the pass that acted on them in a way to invalidate verification because ORAQL will serve cached results later on. The initial four pessimistic queries were reused 265 times later on by

In contrast to sequential version we observed a significant drop in the number of executed instructions, as reported by `perf`, for the optimistically optimized version. With 160 iterations (set via `-ns 160`) the original version executed $934.4 * 10^9$ instructions while the optimistic one only required $860.9 * 10^9$, a reduction of roughly 8%. However, absolute performance was not impacted much, if at all. The 1% improvement we measured is well within the standard derivation of our 11 runs.

*c) TestSNAP - Kokkos - CUDA:* For the TestSNAP Kokkos version we used the CUDA backend and targeted a NVIDIA A100 GPU. We only used ORAQL for the device compilation, thus CPU code was unaffected. All 63302 queries were optimistically answered, 9110 of these were unique. In our experiments we did not observe an impact on the kernel performance. However, for seven out of the 44 kernels generated during the compilation we observed changes in their static properties due to optimistic information. The differences, summarized in Figure 7, affect the number of required registers as well as the stack frame size in bytes. Given rather small scale of the changes and the missing impact on kernel performance we did so far not isolate the root causes for these.

Overall there were 63302 calls to ORAQL during the device code compilation of which 9110, or 14%, were unique.

*d) TestSNAP - Fortran:* Alias analysis has been reported as an important area during the development of "legacy Flang" as well as LLVM/Flang [7]. TestSNAP in particular was used by the LLVM community to perform preliminary performance studies with the "new" LLVM/Flang compiler [8, 9]. An existing experiment already identified aliasing as a performance bottleneck, at least given the LLVM-IR currently emitted by LLVM/Flang (or more precise the "fir-dev" development branch) for a manually modified version of TestSNAP [10]. As the author did not have access to ORAQL, they manually modified LLVM's alias analysis to be optimistic. As full optimism did cause verification problems only "interesting"

| Id | TestSNAP Kokkos - CUDA # registers | TestSNAP Kokkos - CUDA # bytes stack frame | TestSNAP Kokkos - CUDA - ORAQL # registers | TestSNAP Kokkos - CUDA - ORAQL # bytes stack frame | relative difference Δ registers | relative difference Δ bytes stack frame |
|---|---|---|---|---|---|---|
| 1 | 28 | 0 | 32 | 0 | +14.3% | 0% |
| 2 | 28 | 0 | 27 | 0 | -3.7% | 0% |
| 3 | 88 | 56 | 77 | 16 | -14.3% | -71.4% |
| 4 | 56 | 0 | 54 | 0 | -3.7% | 0% |
| 5 | 150 | 56 | 168 | 16 | +10.7% | -71.4% |
| 6 | 86 | 128 | 80 | 88 | -7.5% | -31.3% |
| 7 | 10 | 0 | 8 | 0 | -25% | 0% |

Fig. 7: Impact on the static properties of seven of the 44 kernels generated by the TestSNAP Kokkos targeting CUDA.

translation units were compiled optimistically. They then reported a speedup of $2.14\times$, measured as end-to-end time for a modified input program.

We followed the same steps to build LLVM/Flang (or more precise the "fir-dev" development branch) but then diverted from their experimental setup. For one, we linked all LLVM-IR files obtained without optimization together into a single module. This is effectively a unity or manual LTO build. We then run ORAQL and the fully optimistic version fails to pass verification. This matches the results of the previous experiments. However, ORAQL automatically identifies the 237 unique queries (306 with cached responses) that need to be answered pessimistically.

In our performance evaluation we observed that the optimistically optimized version improves end-to-end time for the `2d_mms_t1.inp` input by 5%. Inspecting the results we noticed that the speedup is located in the setup stage of the proxy app and not the main computation kernel. Hence, the figure of merit for the benchmark is unaffected. In contrast to the existing experiment we did not manually modify the input (which inflates the setup stage artificially) nor did we manually modify the LLVM-IR by hoisting the address computation. As with any transformation, the latter might expose new opportunities for optimistic alias information to make an impact.

### B. XSBench

XSBench [11] is a mini-app that represents the workload of OpenMC, a Monte Carlo neutron application. We applied our ORAQL method to the `Simulation` file as it contains the entire compute kernel. We run three versions of XSBench, sequential, OpenMP, and CUDA. All three show fairly identical behavior in that there is no significant performance difference in the optimistic version and twelve queries are pessimistically answered. In fact, the queries are the same in all versions as they all contain the same constant size array (`dist[12]` in `pick_mat`). One can see that the OpenMP version caused more aliasing queries, which is not surprising given the indirection introduced by any parallelism implementation. The CUDA version is different in that it utilizes the (mostly header-) library NVIDIA Thrust. The large increase in alias queries can be attributed to layers of indirection in that library.

### C. GridMini

GridMini is a substantially reduced version of Grid [12], a C++ lattice Quantum Chromodynamics (QCD) library developed for highly parallel computer architectures. Lattice QCD simulates the strong interactions of quarks and gluons on a four-dimensional discrete space-time grid, and provides crucial input to theoretical nuclear and particle physics.

In our test we run the SU3 benchmark of GridMini with the OpenMP offloading backend on a NVIDIA A100 GPU. To simplify the kernel time comparison we modified the benchmark and evaluated the performance for an `L` value of 60 only. In total, 86 unique alias queries during device-side compilation have been answered optimistically, none were answered pessimistically. The 45 queries originated from Global Value Numbering (GVN), 34 from Memory SSA, three from Dead Store Elimination (DSE), and four from the machine code sinking pass.

While measuring kernel performance we observed a 7% *slowdown* compared to the non-optimistic version. While it is not totally unexpected to see performance degradation originating in additional static information [13], the large drop is surprising. However, it is known that heuristics employed in LLVM are less mature for GPUs. As an example, jump threading is disabled for GPU architectures due to the lack for a reasonable cost function and the realistic chance to significantly degrade performance. The SU3 benchmark is a fairly simple kernel which is why only seven non-alias analysis passes reported statistic outputs with more than 1% difference compared to the non-optimistic run.

### D. Quicksilver

Quicksilver [14] is a proxy app that models the behavior of Mercury [15], which is a code for Monte Carlo transport calculations. The performance of Quicksilver and Mercury is dominated by branching as well as large numbers of small, latency-bound memory loads. Both codes use domain decomposition and node-to-node communication. We experimented with Quicksilver and obtained a fully optimistic version without any pessimistic alias query responses. The runtime is impacted slightly but we do not trust the current results because Quicksilver is unexpectedly and significantly impacted by

the file name of the executable[1]. Without understanding, and hopefully eliminating these performance hazards we decided to withhold runtime numbers for now.

In total, 15 different passes are provided with optimistic alias information, though some, like the memory SSA pass, will further distribute the processed results. With 61%, most optimistic queries were in fact originating from the memory SSA pass [2], followed by Global Value Numbering (GVN) with 18%, loop load elimination with 6.7%, memcpy optimization with 5.5%, and loop invariant code motion with 2.8%. The remaining 5.8% are spread across 10 more passes. The selection of impacted statistics, shown in Figure 6, includes improvements commonly assumed with better alias analysis, e.g., improved load and store elimination as well as increased load hoisting and sinking out of loops. The positive impact on register allocation, namely fewer spilled variables, is however harder to associate as a direct effect. In fact, it might simply result from the deletion of code, e.g., the 53 loops that are now eliminated but not before.

### E. Lulesh

LULESH, the *Livermore Unstructured Lagrange Explicit Shock Hydro* benchmark app, is a simplified test problem to model hydrodynamics workloads [16]. With hundreds of citations, LULESH has been used to evaluate a variety of software analysis or performance optimization and portability techniques, and has been ported to many different programming languages and parallel programming models [17]. We apply ORAQL to the C++ version in its sequential, OpenMP, and MPI variants. In all these versions we focus on the functions that are included in the timed part of the benchmark, which excludes auxiliary functions such as setup and cleanup.

LULESH can not be successfully executed when compiled fully optimistically. ORAQL does however reduce the overall number of no-alias responses by over 60% in all cases, and for the OpenMP variant, almost by a factor of two without changing the displayed result. Even so, the run time is barely affected. For the sequential version of LULESH, the initial and final executables take 18.66s and 18.51s respectively. For OpenMP, the time is 4.18s vs 4.12s, and for MPI (running a larger problem) the time is 47.6s vs 47.7s.

### F. MiniFE

MiniFE [18] is a mini-app from the Mantevo [19] benchmark suite, and is designed to model implicit unstructured finite element solvers. In this paper, we evaluate ORAQL on the "optimized" OpenMP version (`openmp-opt`). We experienced problems running the CUDA version compiled with clang and therefore avoided the GPU ports.

The results of the optimistically optimized OpenMP version are in line with other results we have obtained. The number of additional no-alias results increased slightly, similar to the XSBench OpenMP version. At the same time we found a non-trivial number of alias queries that need to be answered pessimistically. The performance was not impacted but the optimization statistics show the trend observed before, more loads are hoisted and more vectorization is employed.

### G. MiniGMG

MiniGMG [20] is a benchmark that models the code structure of geometric multigrid solvers, which are an important class of linear equation system solvers. We focus on the code versions *ompif* (using OpenMP worksharing loops), *omptask* (using a mix of OpenMP worksharing loops and tasks), and *sse* (using intrinsics for explicit vectorization on SSE instruction sets). For all these versions, we apply ORAQL to the `operators.xxx.c` file, which includes the timed kernel functions of the benchmark.

The compilation instructions of miniGMG[2] are written with the Intel C compiler `icc` in mind, and contain the compiler options `-fno-alias -fno-fnalias`, which are Intel-specific options to globally assume no aliasing. This option essentially acts like ORAQL with a fully optimistic probing sequence, but without allowing fine-grained per-query control.

Since the original compilation instructions for this benchmark assume no aliasing, we expect (and indeed find) that this benchmark passes all tests even with a fully optimistic ORAQL probing sequence. The performance of the SSE version is unaffected by this – we measure a run time of 1.161s for the initial version and 1.157s for the final version. Similarly, the run time of the omptask version reduces from 1.155s by about 1% to 1.144. However, we observe a signficant speedup for the ompif version, which takes 1.299s initially and speeds up to 1.199s, a difference of roughly 8%.

## VI. Lessons Learned

Alias information is always thought of as crucially important for performance. Similarly, the lack of good alias information is often used to justify inefficiencies. While we cannot make a general statement about the role of alias information across all compilers and program domains, we find that it is not the bottleneck (or at least not the only one) for HPC proxy applications compiled with LLVM. Even when parallel programming models are used to utilize multi-cores or accelerators, which generally adds indirections that are hard or impossible for compilers to reason about, there is often no speedup from (almost) perfect alias information.

Since ORAQL is openly available[3] and hopefully soon integrated properly into LLVM, one can expect that future studies explore the effect of aliasing outside the HPC space. While the findings will most likely contain more performance improvements it is already clear that more work has to be done on the transformation side and with regards to analyses of other properties.

[1]We consistently observed a 3× slowdown when the executable was named `qs.final` compared to the default `qs`. We saw similar performance differences between otherwise identical machines. At this time we cannot explain either.

[2]https://crd.lbl.gov/divisions/amcr/computer-science-amcr/par/research/previous-projects/miniGMG/

[3]https://github.com/jhueckelheim/ORAQL

For HPC developers that utilize LLVM directly our results are not necessarily negative. One could argue that LLVM is able to determine all important alias relations already, potentially with the help of annotations, e.g., `restrict`, and command line options, e.g., `-fstrict-aliasing`, both of which we have seen being used in the explored proxy applications. At the same time it is important to note the requirement for pessimistically answered alias queries. Even if they are few in numbers, such queries will always prevent us from ignoring aliasing altogether (like the Intel `-fno-alias` flag does for testing purposes). This is especially true since real applications are often orders of magnitude larger than the proxy apps we have evaluated here.

## VII. Related Work

The ORAQL tool can be seen as an extension of the PETOSPA project [13], which performed optimistic program annotations (including for alias information) combined with a similar probing and testing strategy as in our work. While the spirit is similar, there are differences on the technical side as well as the evaluation and insights gained: In contrast to the PETOSPA equivalent we do not perform explicit modification of the source code (here LLVM-IR code). Instead, ORAQL responds to C++ alias analysis queries as a last resort alias analysis. This distinction is important for two reasons. First, LLVM does not allow to encode fine grained alias information in the IR, hence not all alias queries could be answered optimistically through source annotations. Second, the existing encoding mechanisms for alias information in LLVM-IR are either coarse grained (e.g., the `noalias` attribute PETOSPA employed for argument pointers), or have non-linear scaling behavior. The scaling can easily lead to problems even in relatively small programs [7].

Through our multi-faceted evaluation we can provide insights and comparisons far beyond the reach of related work. We focus especially on the effect of aliasing on parallel programming models as well as other programming abstractions present in the HPC space. However, our openly available tool can be easily deployed in other domains. No related work known to us has done a similar study or provided insights into the difference between programming languages, execution models, and parallelization paradigms when it comes to the impact of aliasing.

The aforementioned previous work on optimistic (or in their words, speculative) alias analysis used dynamic recovery strategies, for example [21, 22]. These works do not consider parallel programs including OpenMP, MPI, CUDA, or Kokkos. Also, they insert recovery strategies to prevent the compiled programs from breaking, which can be useful if these programs are supposed to be used as-is in practice, but due to their run time overhead make it harder to measure potential performance gains that could be achieved by better static analysis or code annotation. Also, these works are not very recent and may not be applicable as-is to modern hardware, e.g., accelerators, or compilers.

A more recent and somewhat similar idea is presented in [23], where the authors propose optimistic static analysis combined with dynamic runtime checks. Unlike the (potentially expensive) recovery strategies in the aforementioned works, the dynamic checks are only guaranteed to catch errors (and cause the program to abort), and the authors demonstrate that there are a a finite number of failure conditions that can trigger aborts, meaning that eventually (after fixing all the bugs revealed by dynamic checks), the program will be correct. Dynamic checks solely for the purpose of detecting but not recovering from bugs has been previously shown within a Fortran research compiler [24].

Other work [25] has observed that alias analysis passes may claim no-alias when aliasing in fact occurs. In their work, they assume that this happens due to bugs in compilers, not because of deliberate speculation as in our work. Still, their proposed method of using run time checks to dynamically detect aliasing may be a useful combination with our work.

Apart from PETOSPA, there is a wealth of literature on identifying the limits of compiler optimizations and the effect of speculatively added or removed information, incl. [26–28].

Alias analysis is an active research field, with recent ideas including finding alias information using deep learning [29], using alias information to exploit undefined behavior for performance gains [30], as a basis for debugging and security analysis of android applications [31], or to make improvements to alias analysis for batch queries [32]. Many other papers make substantial improvements to alias analysis techniques and their use [33–38].

## VIII. Conclusion and Future Work

In our current design we allow existing alias analyses to answer a query first before we fall back to the ORAQL pass. As a consequence we cannot overwrite no-alias and must-alias results derived earlier in the chain. While this does not hinder ORAQL to identify (almost) perfect alias information it disallows us from categorizing the effects of already known queries. We are investigating a design in which we can effectively block existing analyses and provide more pessimistic results in order to determine the effect on subsequent passes and performance.

Future work should determine if optimistic must-alias responses, potentially alone or via a three state deduction, unlocks potential optimizations. Similarly, the combination of optimistic source annotations, as performed by the PETOSPA project, together with (almost) perfect alias information should be explored. When used together, it might be possible to clearly identify the lack of transformations as the main performance bottleneck for HPC applications.

Finally, we expect ORAQL to be used in different domains to determine the impact of insufficient aliasing information. It is reasonable to speculate that programs with more pointer usage and indirection will experience more false positive alias results that actually degrade performance. However, as this study has shown, predictions on the impact of alias information are difficult to make.

## ACKNOWLEDGEMENTS

This work was supported by the Applied Mathematics activity within the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under contract number DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

## REFERENCES

[1] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, 1994. [Online]. Available: https://doi.org/10.1145/186025.186041

[2] D. Novillo *et al.*, "Memory ssa-a unified approach for sparsely representing memory operations," in *Proceedings of the GCC Developers' Summit*. Citeseer, 2007, pp. 97–110.

[3] A. Thompson, C. Trott, S. Plimpton, and USDOE, "Testsnap, version 0.0.1," 11 2019. [Online]. Available: https://www.osti.gov//servlets/purl/1700681

[4] A. P. Thompson, L. P. Swiler, C. R. Trott, S. M. Foiles, and G. J. Tucker, "Spectral neighbor analysis method for automated generation of quantum-accurate interatomic potentials," *J. Comput. Phys.*, vol. 285, pp. 316–330, 2015. [Online]. Available: https://doi.org/10.1016/j.jcp.2014.12.018

[5] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance Portability across Diverse Computer Architectures," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC, P3HPC@SC 2019, Denver, CO, USA, November 22, 2019*. IEEE, 2019, pp. 1–13. [Online]. Available: https://doi.org/10.1109/P3HPC49587.2019.00006

[6] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108171, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465521002836

[7] K. Li and T. Islam, "Towards a representation of arbitrary alias graph in LLVM IR for Fortran code," 2020, lLVM Developers Conference. [Online]. Available: https://llvm.org/devmtg/2020-09/slides/Islam-Li-Fortran_alias_representation.pdf

[8] A. Warzyński, "Building TestSNAP with LLVM/Flang," 2022. [Online]. Available: https://github.com/flang-compiler/f18-llvm-project/issues/1341

[9] M. Petersson, "Performance Analysis for TestSNAP build with LLVM/Flang," 2022. [Online]. Available: https://discourse.llvm.org/t/snap-performance-analysis-more-detailed-than-the-presentation/60636

[10] ——, "Performance Impact of Aliasing on TestSNAP build with LLVM/Flang," 2022. [Online]. Available: https://discourse.llvm.org/t/snap-performance-analysis-more-detailed-than-the-presentation/60636/3

[11] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[12] P. A. Boyle, G. Cossu, A. Yamaguchi, and A. Portelli, "Grid: A Next Generation Data Parallel C++ QCD Library," in *LATTICE*, 2016.

[13] J. Doerfert, B. Homerding, and H. Finkel, "Performance Exploration Through Optimistic Static Program Annotations," in *High Performance Computing - 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proceedings*, ser. Lecture Notes in Computer Science, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds., vol. 11501. Springer, 2019, pp. 247–268. [Online]. Available: https://doi.org/10.1007/978-3-030-20656-7_13

[14] D. F. Richards, R. C. Bleile, P. S. Brantley, S. A. Dawson, M. S. McKinley, and M. J. O'Brien, "Quicksilver: a proxy app for the monte carlo transport code mercury," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 866–873.

[15] R. Procassini, D. Cullen, G. Greenman, and C. Hagmann, "Verification and validation of mercury: a modern, monte carlo particle transport code," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2004.

[16] I. Karlin, J. Keasler, and J. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[17] I. Karlin, "Lulesh programming model and performance ports overview," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2012.

[18] M. Heroux and S. Hammond, "Minife: finite element solver," 2019.

[19] M. Heroux and R. Barrett, "Mantevo project," 2016.

[20] S. Williams, "Implementation and optimization of minigmg-a compact geometric multigrid benchmark," 2012.

[21] M. Fernandez and R. Espasa, "Speculative alias analysis for executable code," in *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*, 2002, pp. 222–231.

[22] W. Ahn, Y. Duan, and J. Torrellas, "Dealiaser: Alias speculation using atomic region support," *SIGPLAN Not.*, vol. 48, no. 4, p. 167–180, mar 2013. [Online].

Available: https://doi.org/10.1145/2499368.2451136

[23] O. Bastani, R. Sharma, L. Clapp, S. Anand, and A. Aiken, "Eventually sound points-to analysis with specifications," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[24] N. Nguyen and F. Irigoin, "Alias verification for fortran code optimization," *Electr. Notes Theor. Comput. Sci.*, vol. 65, pp. 52–66, 04 2002.

[25] J. Wu, G. Hu, Y. Tang, and J. Yang, "Effective dynamic detection of alias analysis errors," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 279–289. [Online]. Available: https://doi.org/10.1145/2491411.2491439

[26] J. Doerfert, T. Grosser, and S. Hack, "Optimistic loop optimization," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. IEEE Press, 2017, p. 292–304.

[27] T. Theodoridis, M. Rigger, and Z. Su, "Finding missed optimizations through the lens of dead code elimination," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 697–709. [Online]. Available: https://doi.org/10.1145/3503222.3507764

[28] S. Siso, W. Armour, and J. Thiyagalingam, "Evaluating auto-vectorizing compilers through objective withdrawal of useful information," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–23, 2019.

[29] J. Eberhardt, S. Steffen, V. Raychev, and M. Vechev, "Unsupervised learning of api aliasing specifications," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 745–759.

[30] A. Phulia, V. Bhagee, and S. Bansal, "Ooelala: Order-of-evaluation based alias analysis for compiler optimization," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 839–853. [Online]. Available: https://doi.org/10.1145/3385412.3385962

[31] K. Ahmed, M. Lis, and J. Rubin, "Mandoline: Dynamic slicing of android applications with trace-based alias analysis," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 105–115.

[32] J. Vedurada and V. K. Nandivada, "Batch alias analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 936–948.

[33] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 32–41. [Online]. Available: https://doi.org/10.1145/237721.237727

[34] A. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-based alias analysis," *SIGPLAN Not.*, vol. 33, no. 5, p. 106–117, may 1998. [Online]. Available: https://doi.org/10.1145/277652.277670

[35] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 242–256. [Online]. Available: https://doi.org/10.1145/178243.178264

[36] M. Shapiro and S. Horwitz, "Fast and accurate flow-insensitive points-to analysis," ser. POPL '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 1–14. [Online]. Available: https://doi.org/10.1145/263699.263703

[37] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," *SIGPLAN Not.*, vol. 42, no. 6, p. 278–289, jun 2007. [Online]. Available: https://doi.org/10.1145/1273442.1250766

[38] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133924