ORAQL — Optimistic Responses to Alias Queries in LLVM

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

Johannes Doerfert[*]
doerfert1@llnl.gov
Argonne National Laboratory
Lemont, IL, USA

ABSTRACT

Alias analysis (AA) is a prerequisite for many compiler optimizations, which are crucial for performance especially for parallel and scientific software. AA is the subject of ongoing research, and compilers can in practice only approximate the alias information of a given program. In this paper we investigate the extent to which performance in high-performance computing (HPC) applications could be improved if better AA were available in LLVM, one of the most widely used compilers today.

To this end we present ORAQL, an optimistic (rather than conservative) AA pass for LLVM that determines AA queries that cannot be answered conclusively by existing techniques, and systematically explores which queries can be answered no-alias without breaking user-provided tests. While ORAQL does not result in provably correct programs and therefore should not be used to compile production code, it allows us to estimate the gap between current and ideal performance. By determining the AA queries that cause the majority of this gap, ORAQL may also guide developers toward beneficial modifications to AA or to HPC programs.

Our results show that the performance of HPC proxy applications across multiple programming languages and parallel programming models is not severely limited by AA when compiled with LLVM, although we show performance gains for some applications.

CCS CONCEPTS

• **Software and its engineering** → **Dynamic compilers**; *Software performance*; • **Mathematics of computing** → *Mathematical software performance.*

KEYWORDS

alias analysis, performance gap estimation, compiler optimization

ACM Reference Format:
Jan Hückelheim and Johannes Doerfert. 2023. ORAQL — Optimistic Responses to Alias Queries in LLVM. In *52nd International Conference on Parallel Processing (ICPP 2023), August 7–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605644

[*]Johannes Doerfert performed this work at Argonne but has since switched affiliation.

1 INTRODUCTION

Alias analysis (AA) is an often cited reason for performance, or the lack thereof. Blaming alias analysis allows one to justify differences in compiler effectiveness, the runtime gap between "low"- and "high"-level languages [28], missing transformations, and other hard-to-explain behaviors. Alias analysis is undecidable [26] and at the same time crucial for performance in some cases. It is thus not surprising that compilers and programming languages have adopted ways to reason about aliasing and the absence of it, such as strict aliasing rules and "restrict" annotations in C/C++, pointer and target attributes in Fortran, the -f[no-]alias command line option in Intel's ICC, and seven alias analyses in LLVM 14 (Basic, ScopedNoAlias, TypeBased, ObjCARC, Globals, CFLAnders, CFLSteens).

In this work we develop a generic methodology to analyze and attribute the effect of alias analysis queries, and we provide a way to estimate the achievable performance with (almost) optimal alias information. We apply this methodology to proxy applications that are routinely used in the literature to represent scientific computing workloads. This allows us to estimate the potential performance gains that could be achieved by more accurate alias analysis for high-performance computing (HPC) applications.

In addition to determining the overall efficiency gap, we can automatically pinpoint the alias queries responsible for the most severe performance degradation in these programs. Further, our approach allows us to measure how different programming languages and parallelization extensions are impacted by alias information. This is particularly interesting since our test set contains C, C++, and Fortran applications, which have different aliasing rules whose effect on performance has been the subject of discussion (see, e.g., [28]).

In contrast to classical alias analyses that attempts to guarantee the absence of aliasing (or dependencies) through static, dynamic, or hybrid reasoning, ORAQL does not yield a sound result for all possible program inputs. Instead, we ensure that the output for a set of user-provided inputs does not change or still satisfies some other user-defined criterion. In some cases, we had to relax the requirement that the output is completely unchanged because roundoff was causing slight fluctuations in the result even without using ORAQL. In these cases we test only the most significant digits in the output that are stable across runs. Hence, our *optimistic alias query responses* are either probably correct or effectively irrelevant for the given set of user inputs, and we cannot reason about inputs not in the set. We note that it is also possible that ORAQL introduces nondeterminism in the program, which may cause the program to succeed in our test for a particular input, then fail for the same input when executed again or on a different system. We have not observed this behavior during our experiments and have consistently found the same final decision sequences and obtained consistent outputs when using the compiled executables afterwards, but there is no guarantee for this in general. 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 locations where potential aliasing degrades performance.

The rest of this paper is structured as follows. In Section 2 the contributions and limitations are clarified before some necessary background is discussed in Section 3. The design of ORAQL is explained in Section 4. An evaluation of ORAQL on seven HPC proxy applications in different configurations is provided in Section 5 followed by a summary in Section 6 of the lessons learned. Related work is discussed in Section 7, and the paper ends in Section 8 with a summary of possible use cases for ORAQL and a brief discussion of future work.

2 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 as follows.

- The ORAQL tool, consisting of an 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.
- A program, generated by ORAQL, compiled with (almost) perfect alias information. Additionally, if requested, ORAQL will generate 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 reimplementations in different programming languages.

ORAQL is a research prototype with limitations, including the following.

- ORAQL is not a program analysis for everyday use. The results are meaningful only for predetermined 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, for example, using cmake.

3 BACKGROUND

The LLVM compiler contains a large number of passes that are called one after another 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 both runtime efficient and memory efficient, they often implement heuristics whose effectiveness depends in nonintuitive 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, 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, for example, memory SSA [21], but its result may or may not impact an actual transformation. In summary, more (optimistic) information does not guarantee any change to the executable, nor do changes imply better performance.

The goal of alias analysis is to determine whether two pointers *alias*—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 because 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 order. A result is returned as soon as a pass in the list 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 list have responded may-alias. By appending the ORAQL alias analysis pass (see Section 4.1) to the end of this list, we ensure that we act only on the alias queries that cannot be analyzed by other passes.

4 APPROACH

ORAQL is implemented as a collaboration of three components: an alias analysis pass within LLVM, a probing driver, and a verification script, the latter two implemented in Python.

4.1 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 decision sequence. The ORAQL pass is appended as the final alias analysis pass in the LLVM pass manager, meaning that ORAQL responds only to those queries that cannot 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 length of decision sequences that need to be probed,

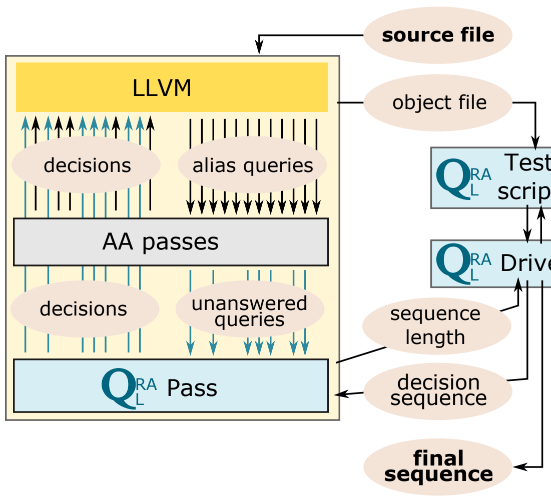

Figure 1: ORAQL workflow. Parts inside the yellow box are within LLVM. Alias queries that cannot be answered by existing passes are given to the ORAQL pass, which answers them based on the decision sequence provided by the driver. The driver repeatedly compiles a file with different decision sequences, until a maximally optimistic decision sequence is found that still allows the compiled executable to satisfy the test criteria when run by the test script.

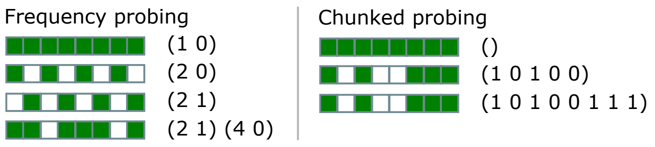

Figure 2: Our two probing strategies. For the frequency strategy, the decision sequence consists of pairs of numbers representing a frequency and a modulo offset. For example, (21) causes optimistic responses for all odd positions in the sequence, whereas (40) would cause optimistic responses for positions $0, 4, 8$, etc. The chunked strategy is instead expressed through a binary sequence where 1 represents an optimistic response. If the query sequence is longer than the decision sequence, subsequent responses are optimistic.

as well as to avoid inconsistencies in our optimistic responses that 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 will fix all optimistic choices after a successful verification. This could theoretically cause us not to find the optimal decision sequence since queries are not independent but can be arbitrarily connected. However, given the large number of optimistic choices that we make in our test cases, and given the fact that our two different probing strategies described below yielded identical final results (though after a different number of probing steps), we believe that probing additional information or changing the decision sequence order is unlikely to yield substantial benefits for the performance of the final executable. On the other hand, it is possible that faster probing strategies could be developed to arrive at the same final executable even faster than our current strategy.

The ORAQL pass is controlled by the probing driver through command line arguments. The most important input is the decision sequence, which is communicated as a series of space-separated "1" (optimistic, no-alias) and "0" (not optimistic, may-alias) characters. The decision sequence is passed through the argument `-opt-aa-seq=<sequence>`. Because decision 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 decision sequence and respond according to that number. If the end of the decision 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 decision sequence length if necessary.

4.2 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 a deactivated ORAQL pass and calls the verification script to ensure a correct compilation and runtime behavior. Then, the driver attempts to answer all queries optimistically by activating the ORAQL pass with an empty decision 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 query sequence length based on the reported number of unique queries from the pass and will 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 1 gives on overview of the collaboration between ORAQL components, as well as their interaction with other LLVM passes, and Figure 3 illustrates recursive probing for a small example.

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 decision 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, as illustrated in Figure 2. Preliminary experiments showed that queries that break the tests if answered optimistically are often clustered together. For this reason, the probing sequence can influence the time it takes to discover a maximally optimistic sequence.

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 a more beneficial query if answering both of them optimistically will break the program. For this reason we refer to the final decision sequence discovered by the probing script as *almost* optimal. Global optimization would require not only checking each query individually but also checking every combination of alias responses.

4.3 ORAQL Verification Script

Before running ORAQL, the user has to obtain a reference output. We did so 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 runtime (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. If desired, a user can replace the call to the regular expression matcher with a custom callback function that uses other criteria to check the validity of the output, for example to check whether some number in the output is within a certain range. We note that our tests use only 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.

4.4 Static Impact Identification

Many programs can be successfully verified with significant optimistic knowledge. Therefore, 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 whether initial or cached queries should be printed, while the flags `-opt-aa-dump-{optimistic,pessimistic}` determine whether 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

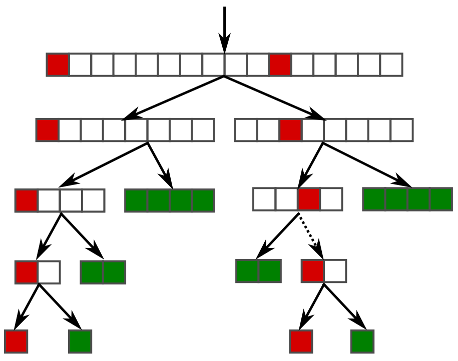

Figure 3: Probing in ORAQL driver. The top row represents unanswered queries passed to ORAQL. Most of them, represented by white squares, can be answered optimistically without failed tests, while a few queries, represented by red squares, cannot. ORAQL recursively refines the responses to pinpoint those "dangerous" queries. Dotted arrows represent outcomes that were deduced from nearby queries.

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 4 shows all four pessimistically answered non-cached queries for the TestSNAP OpenMP version (see Section 5.1), 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 the printing of pass names prior to their execution with the `-debug-pass=Executions` flag. Note that because of 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 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.

4.5 Multitarget Compilation

The prevailing compilation model for accelerators uses a single source file containing host and device code. That usually means compilers run multiple times on a source file, once per target architecture. To simplify the use of ORAQL for offloading (e.g., CUDA and OpenMP offload), we introduced the command line flag `-opt-aa-target=<target-sub-string>`, which restricts ORAQL to a given architecture. If ORAQL is used for both parts, we currently cannot adjust the decision sequence between the different compilations run of the same file. Effectively, ORAQL will reuse the decision sequence for all targets, which will lead to a pessimistic intersection of all of them.

5 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

```
[...] Executing Pass 'Global Value Numbering' on Function '.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
```

Figure 4: ORAQL debug output, during the compilation with OpenMP enabled, of the main source file of TestSNAP (see Section 5.1), showing a pessimistically answered queries. The first line shows the pass that issued the queries, here global value numbering (GVN). The following lines show response kind, cache status, the two pointers and the location description, and other information if available such as containing function or line number of pointer origination.

			# Opt. Queries		# Pess. Queries		# No-Alias Results		
Benchmark	Programming Model	Source Files	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%

Table 1: Statistics for all benchmarks. The *# Opt.* and *# Pess.* columns count the optimistic or pessimistic answers in the final decision sequence, respectively. The right columns count the overall no-alias responses across all passes (not just ORAQL).

offload runs. The former were performed on Intel(R) Xeon(R) Platinum 8180M CPU @ 2.50 GHz (Skylake) CPUs with 112 threads and 768 GB of memory. The latter were executed on an NVIDIA A100 GPU. Depending on the experiment, we report the number of executed instructions, wall-clock time, figure of merit results, or GPU kernel times. The software versions used are LLVM, LLVM/Flang (fir-dev), and Legacy Flang at Git commits ea7be7e, 972e1f8, and b90b722, CUDA 11.4.0, and Kokkos 3.5.0. The benchmarks are briefly summarized in Table 1.

In the following we introduce the benchmarks and discuss their performance with and without ORAQL. The results are also summarized in Table 4 and in Section 6. 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 since 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 benchmarks or compiler.

5.1 TestSNAP

TestSNAP is a proxy for the SNAP force calculation in the LAMMPS molecular dynamics package [6, 33–35]. It contains synthetic inputs (atom positions) and reference outputs (forces on atoms) for different SNAP models and performs a force calculation before reporting grind time (msec/atomstep) and RMS force error (eV/A).

We used four versions of TestSNAP for our experiments not only to show the applicability of ORAQL but also to 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 an NVIDIA A100 GPU. For all versions we run ORAQL only on the sna.cpp file that contains the main computation kernel. The fourth, a Fortran version, is compiled with LLVM/Flang from 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

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%

Table 2: Selection of statistics reported by `-mllvm -stats` for the original and ORAQL compilations.

linked them manually into a single LLVM-IR bitcode file that we optimized with ORAQL as part of the `-O3` optimization pipeline.

TestSNAP – Sequential. The sequential version of TestSNAP was compiled fully optimistic without invalidating our verification constraints. In total, 68,177 queries were answered as optimistic no-alias, and 44% of these were unique. In the original version only 44,259 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 Table 2. The number of executed instructions, as reported by `perf`, was down by 1.2%. The performance improved by 3.6%, with both the `-DREFDATA_TWOJ =14100` build parameter and the `-ns 10` command line option.

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` that 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 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.

In contrast to the 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 required $860.9*10^9$, a reduction of roughly 8%. However, the absolute performance improvement of 1% is within the standard derivation of our 11 runs.

TestSNAP – Kokkos – CUDA For the TestSNAP Kokkos version we used the CUDA backend and targeted an NVIDIA A100 GPU. We used ORAQL only for the device compilation; thus CPU code was unaffected. All 63,302 queries were optimistically answered; 9,110 of these were unique. In our experiments we did not observe an impact on the kernel performance. However, for 7 out of the 44 kernels generated during the compilation, we observed changes in their static properties due to optimistic information. The differences, summarized in Table 3, affect the number of required registers as well as the stack frame size in bytes. Given the rather small scale of the changes and the missing impact on kernel performance, so far we have not isolated the root causes for these.

TestSNAP – Fortran Alias analysis has been reported as an important area during the development of "legacy Flang" as well as LLVM/Flang [19]. TestSNAP in particular was used by the LLVM community to perform preliminary performance studies with the "new" LLVM/Flang compiler [22, 38]. An existing experiment already identified aliasing as a performance bottleneck, at least given the LLVM-IR currently emitted by LLVM/Flang (or, more precisely, the "fir-dev" development branch) for a manually modified version of TestSNAP [23]. Since the authors 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" translation units were compiled optimistically, resulting in a reported 2.14× speedup for a modified input program.

	TestSNAP Kokkos – CUDA		**TestSNAP Kokkos – CUDA – ORAQL**		**Relative difference**	
Id	**# registers**	**# bytes stack frame**	**# registers**	**# bytes stack frame**	**Δ registers**	**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%

Table 3: Impact on the static properties of 7 of the 44 kernels generated by the TestSNAP Kokkos targeting CUDA.

We followed the same steps to build LLVM/Flang (or, more precisely, 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 ran ORAQL and found that the fully optimistic version fails verification, but we determined that 237 unique queries (306 with cached responses) need to be answered pessimistically to succeed in tests.

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.

5.2 XSBench

XSBench [36] is a mini-app that represents the workload of OpenMC, a Monte Carlo neutron application. We applied our ORAQL method to the `Simulation` file since it contains the entire compute kernel. We ran three versions of XSBench—sequential, OpenMP, and CUDA—all of which require the same twelve pessimistically answered queries. ORAQL does not result in significant performance differences in the versions. 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.

5.3 GridMini

GridMini is a simplified version of Grid [4], 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 insights in nuclear and particle physics.

In our test we ran the SU3 benchmark of GridMini with the OpenMP offloading backend on an 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 were answered optimistically; none were answered pessimistically. The 45 queries originated from 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 with the non-optimistic version. While it is not totally unexpected to see performance degradation originating in additional static information [9], 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 because of the lack of 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 with the non-optimistic run.

5.4 Quicksilver

Quicksilver [27] is a proxy app that models the behavior of Mercury [25], 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]

In total, 15 different passes are provided with optimistic alias information, although 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 [21], followed by 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 Table 2, includes improvements commonly assumed with better alias analysis, for example, 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 harder to associate as a direct

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

effect. In fact, it might simply result from the deletion of code, for example, the 53 loops that are now eliminated.

5.5 LULESH

LULESH, the *Livermore Unstructured Lagrange Explicit Shock Hydro* benchmark app, is a simplified test problem to model hydrodynamics workloads [17]. 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 [16]. 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 cannot 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 2 without changing the displayed result. Even so, the runtime is barely affected. For the sequential version of LULESH, the initial and final executables take 18.66 s and 18.51 s, respectively. For OpenMP, the time is 4.18 s vs 4.12 s, and for MPI (running a larger problem) the time is 47.6 s vs 47.7 s.

5.6 MiniFE

MiniFE [14] is a mini-app from the Mantevo [13] 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 nontrivial 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.

5.7 MiniGMG

MiniGMG [39] 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 decision sequence, but without allowing fine-grained per-query control.

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

Benchmark	Programming Model	ORAQL Speedup
TestSNAP	C++	**3.6%**
TestSNAP	C++, OpenMP	**1.0%**
TestSNAP	C++, Kokkos, CUDA	**0**
TestSNAP	Fortran	**5%**
XSBench	all versions	**0**
GridMini	C++, OpenMP Offload	**-7%**
Quicksilver	C++, OpenMP	*[inconclusive]*
LULESH	C++	**0.8%**
LULESH	C++, OpenMP	**1.4%**
LULESH	C++, MPI	**-0.2%**
MiniFE	C++, OpenMP	**0**
MiniGMG	C, SSE intrinsics	**3.4%**
MiniGMG	C, OpenMP tasks	**1%**
MiniGMG	C, OpenMP	**8.3%**

Table 4: Speedup or slowdown obtained with ORAQL. Most test cases show a modest performance improvement, with a maximum of over 8% for MiniGMG. The GridMini test case shows a 7% performance degradation. The Quicksilver timing results are inconclusive, due to excessive runtime fluctuations in our experiments regardless of ORAQL usage.

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 decision sequence. We measure a runtime of 1.161 s for the initial SSE version and 1.157 s after applying ORAQL, a 3% improvement. The runtime of the omptask version reduces slightly, from 1.155 s by about 1% to 1.144 s. However, we observe a significant speedup for the ompif version, which takes 1.299 s initially and speeds up to 1.199 s, a difference of roughly 8%.

6 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 multicores or accelerators, which generally add indirections that are hard or impossible for compilers to reason about, we found little speedup from (almost) perfect alias information in our test cases. In other words, performance barely improves compared with the state of the art even if all memory reference pairs are marked correctly as must-alias or no-alias.

Since ORAQL is openly available[3] and (we hope) soon integrated properly into LLVM, we expect that future studies can build on this method to explore the effect of aliasing for applications other than HPC proxy apps. Our study may also point at other transformations or analyses that need to be developed in order to actually benefit from better alias information, if it was available.

[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, for example, `restrict`, and command line options, for example, `-fstrict-aliasing`, both of which we have seen being used in the explored proxy applications. At the same time we note the requirement for pessimistically answered alias queries. Even if they are few in number, 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.

7 RELATED WORK

ORAQL can be seen as an extension of the PETOSPA project [9], which performed optimistic program annotations (including for alias information) combined with probing and testing. There are technical differences, and different insights were 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 allows us to encode fine-grained alias information in the IR, which is not possible through source annotations. Moreover, the existing encoding mechanisms for alias information in LLVM-IR either are coarse grained (e.g., the `noalias` attribute PETOSPA employed for argument pointers) or have nonlinear scaling behavior. The scaling can lead to problems even in relatively small programs [19].

Also, unlike previous work, we focus on the effect of aliasing on parallel programming models, on different execution models, and across languages, as well as programming abstractions present in HPC, even though the tool could be deployed in other domains.

The previous works on optimistic (or, in their words, speculative) alias analysis used dynamic recovery strategies, for example [2, 12]. 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 are useful for running the programs in practice but have a runtime overhead that makes it hard to measure potential performance gains that could be achieved by better static analysis or code annotation. Also, these works are not recent and did not consider accelerators.

A more recent and somewhat similar idea is presented in [3], 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 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 [20]. Other work has discussed optimizations enabled by runtime alias checks targeted at locations whose alias status cannot be statically determined [5].

Other work [40] has observed that alias analysis passes may claim no-alias when aliasing in fact occurs. In their work, the authors assume that this situation happens due to bugs in compilers, not because of deliberate speculation as in our work. Still, their proposed method of using runtime 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, including [8, 30, 32].

Alias analysis is an active research field, with recent ideas including finding alias information using deep learning [10], using alias information to exploit undefined behavior for performance gains [24], using alias analysis as a basis for debugging and security analysis of android applications [1], or making improvements to alias analysis for batch queries [37]. Many other papers make substantial improvements to alias analysis techniques and their use [7, 11, 15, 18, 29, 31].

8 CONCLUSION AND FUTURE WORK

Beyond the applications in this paper, we believe that there are various use cases for ORAQL, including the following:

Sporadic source code tuning. Developers who modify and annotate their program can achieve better performance. In contrast to marking all arguments as “restrict," the ORAQL workflow minimizes changes to the program while retaining most benefits, which is useful because any annotation induces maintenance cost—for example, the invariant has to be preserved over time.

Compiler development. ORAQL can be used to identify 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 that of generic alternatives. Since the most costly alias analysis are often disabled by default because of their scaling behavior, specialized versions that cover the most important cases provide a new trade-off not available so far.

Bounded search space for alias analysis-related tuning techniques. 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 hyperparameters was in practice done by hand. A bounded search space allows stopping if the improvement potential becomes negligible.

In our current design we allow existing alias analyses to answer a query first before we fall back to the ORAQL pass. Consequently, we cannot overwrite no-alias and must-alias results derived earlier in the chain. While this does not prevent ORAQL from identifying (almost) perfect alias information, it prevents 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 whether ORAQL combined with optimistic source annotations or optimistic *must-alias* responses can unlock performance gains or help confirm the lack of transformations as the main performance bottleneck for HPC applications. Furthermore, we expect ORAQL to be used in different domains where the impact of insufficient aliasing information could be larger because of more frequent use of indirections and pointers.

ACKNOWLEDGMENTS

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] Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. Mandoline: Dynamic Slicing of Android Applications with Trace-Based Alias Analysis. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 105–115. https://doi.org/10.1109/ICST49551.2021.00022

[2] Wonsun Ahn, Yuelu Duan, and Josep Torrellas. 2013. DeAliaser: Alias Speculation Using Atomic Region Support. *SIGPLAN Not.* 48, 4 (mar 2013), 167––180. https://doi.org/10.1145/2499368.2451136

[3] Osbert Bastani, Rahul Sharma, Lazaro Clapp, Saswat Anand, and Alex Aiken. 2019. Eventually sound points-to analysis with specifications. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[4] Peter A. Boyle, Guido Cossu, Azusa Yamaguchi, and Antonin Portelli. 2016. Grid: A Next Generation Data Parallel C++ QCD Library. In *LATTICE*.

[5] Khushboo Chitre, Piyus Kedia, and Rahul Purandare. 2022. The Road Not Taken: Exploring Alias Analysis Based Optimizations Missed by the Compiler. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 153 (oct 2022), 25 pages. https://doi.org/10.1145/3563316

[6] Tom Deakin, Simon McIntosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. 2019. 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, 1–13. https://doi.org/10.1109/P3HPC49587.2019.00006

[7] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-Based Alias Analysis. *SIGPLAN Not.* 33, 5 (may 1998), 106––117. https://doi.org/10.1145/277652.277670

[8] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. 2017. Optimistic Loop Optimization. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 292–304.

[9] Johannes Doerfert, Brian Homerding, and Hal Finkel. 2019. 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 (Lecture Notes in Computer Science, Vol. 11501)*, Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan (Eds.). Springer, 247–268. https://doi.org/10.1007/978-3-030-20656-7_13

[10] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. 2019. Unsupervised learning of API aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 745–759.

[11] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. 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* (Orlando, Florida, USA) *(PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 242—-256. https://doi.org/10.1145/178243.178264

[12] M. Fernandez and R. Espasa. 2002. Speculative alias analysis for executable code. In *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*. 222–231. https://doi.org/10.1109/PACT.2002.1106020

[13] Michael Heroux and Richard Barrett. 2016. Mantevo project.

[14] Mike Heroux and Simon Hammond. 2019. MiniFE: finite element solver.

[15] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-Driven Context-Sensitivity for Points-to Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (October 2017), 28 pages. https://doi.org/10.1145/3133924

[16] Ian Karlin. 2012. *Lulesh programming model and performance ports overview*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[17] Ian Karlin, Jeff Keasler, and JR Neely. 2013. *Lulesh 2.0 updates and changes*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[18] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World. *SIGPLAN Not.* 42, 6 (June 2007), 278––289. https://doi.org/10.1145/1273442.1250766

[19] Kelvin Li and Tariqu Islam. 2020. Towards a representation of arbitrary alias graph in LLVM IR for Fortran code. https://llvm.org/devmtg/2020-09/slides/Islam-Li-Fortran_alias_representation.pdf LLVM Developers Conference.

[20] Nga Nguyen and François Irigoin. 2002. Alias Verification for Fortran Code Optimization. *Electr. Notes Theor. Comput. Sci.* 65 (04 2002), 52–66. https://doi.org/10.1016/S1571-0661(04)80396-7

[21] Diego Novillo et al. 2007. Memory SSA – a unified approach for sparsely representing memory operations. In *Proceedings of the GCC Developers' Summit*. Citeseer, 97–110.

[22] Mats Petersson. 2022. Performance analysis for TestSNAP build with LLVM/Flang. https://discourse.llvm.org/t/snap-performance-analysis-more-detailed-than-the-presentation/60636

[23] Mats Petersson. 2022. Performance Impact of Aliasing on TestSNAP build with LLVM/Flang. https://discourse.llvm.org/t/snap-performance-analysis-more-detailed-than-the-presentation/60636/3

[24] Ankush Phulia, Vaibhav Bhagee, and Sorav Bansal. 2020. OOElala: Order-of-Evaluation Based Alias Analysis for Compiler Optimization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 839––853. https://doi.org/10.1145/3385412.3385962

[25] RJ Procassini, DE Cullen, GM Greenman, and CA Hagmann. 2004. *Verification and validation of Mercury: A modern, Monte Carlo particle transport code*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[26] G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1467–1471. https://doi.org/10.1145/186025.186041

[27] David F Richards, Ryan C Bleile, Patrick S Brantley, Shawn A Dawson, Michael Scott McKinley, and Matthew J O'Brien. 2017. Quicksilver: a proxy app for the Monte Carlo transport code mercury. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 866–873.

[28] Michael Lee Scott. 2006. *Programming language pragmatics* (2nd ed.). Morgan Kaufmann.

[29] Marc Shapiro and Susan Horwitz. 1997. Fast and Accurate Flow-Insensitive Points-to Analysis *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 1––14. https://doi.org/10.1145/263699.263703

[30] Sergi Siso, Wes Armour, and Jeyarajan Thiyagalingam. 2019. Evaluating auto-vectorizing compilers through objective withdrawal of useful information. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–23.

[31] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32––41. https://doi.org/10.1145/237721.237727

[32] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. 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* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 697–709. https://doi.org/10.1145/3503222.3507764

[33] Aidan Thompson, Christian Trott, Steven Plimpton, and USDOE. 2019. TestSNAP, Version 0.0.1. https://doi.org/10.11578/dc.20201030.5

[34] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. 2022. LAMMPS – a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications* 271 (2022), 108171. https://doi.org/10.1016/j.cpc.2021.108171

[35] Aidan P. Thompson, Laura P. Swiler, Christian R. Trott, S. M. Foiles, and Garritt J. Tucker. 2015. Spectral neighbor analysis method for automated generation of quantum-accurate interatomic potentials. *J. Comput. Phys.* 285 (2015), 316–330. https://doi.org/10.1016/j.jcp.2014.12.018

[36] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. 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).

[37] Jyothi Vedurada and V. Krishna Nandivada. 2019. Batch Alias Analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 936–948. https://doi.org/10.1109/ASE.2019.00091

[38] Andrzej Warzyński. 2022. Building TestSNAP with LLVM/Flang. https://github.com/flang-compiler/f18-llvm-project/issues/1341

[39] Samuel Williams. 2012. Implementation and optimization of miniGMG – a compact geometric multigrid benchmark. (2012).

[40] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. 2013. Effective Dynamic Detection of Alias Analysis Errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 279–289. https://doi.org/10.1145/2491411.2491439