Performance Exploration Through Optimistic Static Program Annotations

Johannes Doerfert[0000−0001−7870−8963], Brian Homerding[0000−0002−5455−6181], and Hal Finkel[0000−0002−7551−7122]

Argonne National Laboratory, Lemont, IL, USA
{jdoerfert,bhomerding,hfinkel}@anl.gov

Abstract. Compilers are limited by the static information directly or indirectly encoded in the program. Low-level languages, such as C/C++, are considered problematic as their weak type system and relaxed memory semantic allows for various, sometimes non-obvious, behaviors. Since compilers have to preserve the program semantics for all program executions, the existence of exceptional behavior can prevent optimizations that the developer would consider valid and might expect. Analyses to guarantee the absence of disruptive and unlikely situations are consequently an indispensable part of an optimizing compiler. However, such analyses have to be approximative and limited in scope as global and exact solutions are infeasible for any non-trivial program.

In this paper, we present an automated tool to measure the effect missing static information has on the optimizations applied to a given program. The approach generates an optimistically optimized program version which, compared to the original, defines a performance gap that can be closed by better compiler analyses and selective static program annotations.

Our evaluation on six already tuned proxy applications for high-performance codes shows speedups of up to 20.6%. This clearly indicates that static uncertainty limits performance. At the same time, we observed that compilers are often unable to utilize additional static information. Thus, manual annotation of all correct static information is therefore not only error prone but also mostly redundant.

Keywords: Compiler Guided Auto Tuning · Performance Gap · LLVM

1 Introduction

Programs in the high-performance computing domain are often subject to fine-grained tuning and therefore developed in low-level programming languages such as Fortran or C/C++. However, this tuning potential can cut both ways. Without proper annotations, low-level languages allow various behaviors that are uncommon to occur during a normal program execution. These "corner case behaviors" include, for example, potentially aliasing pointers and possibly overflowing integer operations. While performance can increase if such corner case behaviors are exploited properly, performance can also be limited if beneficial compiler transformations are prevented by their presence.

Figure 1 illustrates how corner case behaviors can prevent desired optimizations. The call to the `external` function might cause arbitrary side effects and changes to the values passed as arguments. After the call, `sum` might not be zero and `locP` is not guaranteed to be `{5, 11}`. Additionally, the address of `sum` or `locP` could escape, creating aliasing issues if one is stored in `globalPtr`. As a result, compilers cannot assume the access to `globalPtr` is invariant in the loop. Finally, the loop iteration counter `u` may overflow. Thus, the loop can iterate either `UB - LB` iterations, if `LB <= UB`, or alternatively `256 - UB + LB` iterations, if `LB > UB`. Due to this uncertainty, most compilers will struggle to optimize the loop, e.g., to replace it by a closed form expression. As discussed in Section 2, all these optimizations would be possible *if* better static information on the effects of the `external` function and the values of `LB` and `UB` were available.

```
int *globalPtr;
void external(int*, std::pair<int>&);

int foo(uint8_t LB, uint8_t UB) {
  int sum = 0;
  std::pair<int> locP = {5, 11};
  external(&sum, locP);
  for (uint8_t u = LB; u != UB; u++)
    sum += *globalPtr + locP.first;
  return sum;
}
```

Fig. 1: Low-level code that allows for various unexpected behaviors which prevent performance critical transformations.

In this work we identify and optimistically eliminate situations in which static information is missing, e.g., due to the low-level nature of the program. In particular, we determined 20 opportunities for which skilled, performance-minded developers, or improved compiler analyses, could enhance conservatively sound compiler assumptions. For these, our tool automatically explore the performance impact if perfect information would have been provided by optimistically providing it, followed by an application specific verification step. In other words, we automatically accumulate optimistic static program annotations under which the program remains valid on user specified inputs. After this tuning process, the last successfully verified version defines a performance gap which can be minimized through manual annotations.

It is important to note that optimistic optimization is not meant to be used in production because it gives up on total correctness, the foundation of the compilation process. Instead, it should be seen as a compiler guided development tool. It directs performance minded programmers towards static information both *required and usable* by the compiler, consequently minimizing manual effort while effectively increasing performance.

The paper is organized as follows: Section 2 explains how static annotations restrict the set of defined program behaviors, potentially enabling program optimizations. In Section 3, we detail the exploited opportunities for additional static information. We explain the corner case behaviors which annotation can exclude, the transformations that could be enabled, and how static information can be provided in the source. Before we present an elaborate evaluation of our approach on six high-performance proxy applications in Section 5, we list implementation choices in Section 4. After related work is discussed in Section 6, we conclude in Section 7.

2 Static Program Annotation

Programming languages, including the intermediate representations used inside a compiler, allow to encode additional information directly in the program code. Such information can improve later analyses and enable optimizations, regardless of how the information came to be. This is especially important for performance aware developers that use program annotations to encode their domain knowledge, e.g., the shapes of possible inputs or the contexts in which code will be used. Encoded information lifts a burden from the compiler as it limits the set of defined program behaviors. The only (purely static) alternative is running complex analyses at compile

```
int *globalPtr;
void external(int*, std::pair<int>&)
     __attribute__((pure));
int foo(uint8_t LB, uint8_t UB) {
  int sum = 0;
  std::pair<int> locP = {5, 11};
  external(&sum, locP);
  __builtin_assume(LB <= UB);
  for (uint8_t u = LB; u != UB; u++)
    sum += *globalPtr + locP.first;
  return sum;
}
```

```
int *globalPtr;
void external(int*,std::pair<int>&);

int foo(uint8_t LB, uint8_t UB) {
  int sum = 0;
  std::pair<int> locP = {5, 11};
  external(&sum, locP);


  int gPVal = *globalPtr;
  return (UB - LB) * (gPVal+5);
}
```

Fig. 2: Left: The code shown in Figure 1 statically annotated with optimistic information. Right: The same code after annotation enabled compiler optimizations eliminated the loop.

time. Given that some program properties, e.g., pointer aliasing, are in their general form undecidable [20], programmer annotated knowledge is often irreplaceable.

Section 1 lists problems and missed transformations for the code in Figure 1. To overcome these problems, and thereby enable optimizations that lead to the version shown in Figure 2 (right), the problematic corner case behaviors need to be eliminated through program annotations. The first was the potential for the `external` function call to manipulate the arguments as well as `globalPtr`. While our implementation in the LLVM [16] compiler can encode this in different ways, programming languages like C/C++ generally offer less possibilities. In this situation it is *sufficient* to annotate the `external` function as `pure`, as shown in the left part of Figure 2. Pure functions may not alter outside state, preventing the escape of the argument pointers and thereby also potentially aliasing accesses. Consequently, the compiler can hoist the load of `globalPtr` out of the loop. Also the access to `locP.first` can be hoisted which leaves a loop that accumulates an unknown but *fixed* value. If we additionally ensure the compiler that the loop iteration variable `u` is not going to wrap around, e.g., through the `__builtin_assume(LB <= UB)` annotation, the loop is replaced by a closed form expression that directly computes the result. Note that the absence of memory writes in the external function, the consequent absence of aliasing pointers, and the lifetime of `sum` allow to simplify the closed form expression even further.

As manual exploration of such annotations is tedious, and most are superfluous, we provide an automated way. It allows developers to periodically determine the impact of some, or all, static uncertainty sources in application hotspots and, depending on the results, manually verify and manifest the most important annotations in the code.

3 Optimistic Optimization Opportunities

Optimistic optimization opportunities arise whenever the semantic of the program allows different behaviors to manifest at runtime. While this is the essence of any input-dependent, non-trivial program, there are various situations for which the runtime behavior for all inputs, or at least the ones the user is interested in, is actually the same. While the purpose of compiler analyses is to identify which behaviors cannot occur at runtime, optimistic optimization opportunities allow to explore the space of the ones we need to allow. Thus, program analyses find a potentially conservative, but sound approximation of the actual runtime behaviors while optimistic optimization opportunities enable us to explore less conservative, potentially unsound approximations. Figure 3 lists the optimistic optimization opportunity kinds our approach can identify and exploit. Whenever one of these situations is encountered in the program, our compiler extension generates an optimistic choice, which, if taken, results in a program annotation that limits the behaviors the rest of the compiler will assume to be legal.

In the remainder of this section, identified and exploited opportunity kinds and their source annotations are detailed together with a discussion how subsequent transformations may be enabled by seized optimistic opportunities.

Description	**Section**
pot. overflowing computations	3.1
pot. parallel loops	3.2
unknown control flow choices	3.3
pot undefined behavior in functions	3.4.1
unknown function side-effects	3.4.2
pot runtime exceptions in functions	3.4.3
unknown function return values	3.4.4
externally visible functions	3.4.5
pot. aliasing pointers	3.5.1
pot. escaping pointers	3.5.2
unknown pointer usage	3.5.3
unknown pointer alignment	3.5.4
pot. non-dereferenceable pointers	3.5.5
pot. invariant memory locations	3.5.6

Fig. 3: Identified and exploited opportunities.

3.1 Potentially Overflowing Computations

Binary computations in low-level languages, such as C/C++ and LLVM intermediate representation (LLVM-IR), have multiple evaluation semantics that differ for overflowing operations. In C/C++, the signedness of the operands determines if operations are computed with *wrapping* or *undefined* semantics. For the former, the result of the operation is computed modulo the largest value representable in the target type bit-width. For the latter, the value is undefined if the mathematically exact result would require more bits than provided by the target type. In LLVM-IR, values do not have an associated signedness but operations carry annotations to determine the semantics. If none are present, *wrapping* semantic is used. While it is an implementation of *undefinied* semantic, it is more restrictive when it comes to the possibility of integer overflows. Only if operations are tagged with no-(un)signed-wrap (`nsw`/`nuw`), LLVM is allowed to assume the more lenient *undefinied* semantic. Thus, the result, if interpreted as signed or unsigned value respectively, will either be mathematically exact or undefined. Similarly, C/C++ compilers allow to enforce *wrapping* or *undefined* semantic for potentially overflowing computations regardless of the operands signedness through the command line options `-fwrapv` and `-fno-strict-overflow`.

Potentially overflowing arithmetic operations, including address computations, offer an opportunity for optimistic annotation. If the program semantic did not imply `nsw`/`nuw` for a computation, or if transformations applied by the compiler could not prove these properties for newly introduced or modified code, a potential overflow is well-defined and has to be taken into consideration. As integer overflows rarely happen (on purpose) in practice, especially in loop heavy computation hot-spots [1,9], the missing `nsw`/`nuw` flags provide a perfect opportunity for optimistic optimization.

Our optimistic code annotator can add missing annotations, e.g., `nsw` and `nuw`, to potentially overflowing operations. We distinguish thereby between annotations for signed, unsigned, and address computations which can be enabled separately.

3.2 Potentially Parallel Loops

Detecting parallelism in sequential programs has been a major challenge for decades. Especially for low-level languages there are various caveats including, but not limited to: potentially overflowing computations (ref. Section 3.1), potentially aliasing pointers (ref. Section 3.5.1), and unknown side-effects of function calls (ref. Section 3.4.2). Even if all of these issues are tamed, powerful dependence analyses are needed to identify parallelism in non-trivial loops [11,19,5].

In the context of LLVM, parallelism is usually exploited by the loop vectorizer and the polyhedral loop optimizer Polly [13]. While both employ runtime checks to deal with some of the aforementioned low-level issues [3,9], these come with their own set of limitations. As our approach shifts the soundness liability to the expert developer, we can optimistically annotate loops as parallel.

LLVM currently encodes parallelism as metadata annotations on non-pure instructions inside of loops. The annotations are only exploited in two ways, both related to the loop vectorizer: First, the dependency legality check for vectorization is skipped, and, second, in case if-conversion [2] is necessary, it is assumed to be legal.

3.3 Control Flow Speculation

Programs, especially in high-performance computing, often interleave various operating modes that result in variations in the executed program path. In the benchmarks we evaluated, input flags determined for example which energy transfer function and output method is used. In case we are only interested in a subset of these modes, we can specialize the program based on the content of variables which determine the executed path. Thus, if a variable is used as a control condition, we can optimistically assume that only one control flow target is always executed next. To embed this information in the program, we place an assumptions intrinsics call (`llvm.assume`) which is LLVM's counterpart of Clang's `__builtin_assume`. Other C/C++ compilers have similar functionality.

Similar to general value specialization, which could also be done through this scheme, unguided control flow speculation is unlikely to succeed. We therefore restrict ourselves to the control flow conditions that depend on global variables, parameters, and function return values. Additionally, we do not speculate for loop exit or latch branches, and we require a non-relational control flow condition with one constant operand. While this already reduces the possibilities significantly, we additionally try to use a single optimistic optimization choice variable to represent all opportunities induced by the same a global variable, function return value, or function parameter. This will synchronize all speculative choices as described in Section 4.1.

3.4 Function Behavior

A compiler has to treat calls to unknown functions as optimization barriers because the callee can not only cause arbitrary side-effects, but it could also never return control to the caller. Even if the called function is known, its definition might not necessarily be available in the current translation unit. If a definition is available but the language semantic allows a different one to be chosen at link time or run-time, it is not allowed to deduce information from this *potential* definition. Finally, if the definition is available and known to be executed, the compiler has to employ inter-procedural analyses. From an algorithmic standpoint such inter-procedural analyses are often less precise, due to uncertainty stemming from unknown outside callers. From an implementation standpoint they are also less interesting than their intra-procedural counterparts because the latter are predominantly needed after (aggressive) inlining was performed.

In LLVM, intra-procedural analyses are dominating in numbers and potential. The existing inter-procedural analyses mostly try to limit the possible effects of function calls and simplify the caller-callee interface through propagation of constants. However, all of the above mentioned issues will limit the information that can be deduced from, and the transformations than can be applied to, functions.

Since function call can generally cause various possible behaviors at runtime, especially if the called function is unknown or not inlined, they provide different optimistic optimization opportunities discussed in the following.

3.4.1 Undefined Behavior Functions might not only cause side-effects and raise exceptions, they can also cause undefined behavior, e.g., a division by zero. While compilers generally take advantage of undefined behavior, they shall never introduce it on a path on which it would not manifest anyway. Consequently, unconditionally hoisting of calls out of a loop is unsound, even if the call is to a *constant* function (ref. Section 3.4.2) not raising exceptions (ref. Section 3.4.3). Doing so is only valid if the callee does either not cause undefined behavior, or it would have been executed anyway.

To enable control dependence changes for calls, we provide an optimistic optimization opportunity for the `speculatable` LLVM-IR function attribute. Since `speculatable` does imply the absence of undefined behavior and also other side-effects, we combined this opportunity with the side-effect encoding described in Section 3.4.2.

3.4.2 Side-Effects Conservatively, a function might read or write any accessible memory location. Thus, everything transitively reachable through global variables or pointer arguments is potentially accessed. Since this generally includes locations to which pointers might have escaped earlier (ref. Section 3.5.2), the set of *known invariant locations* is often quite limited. Consequently, transformations involving memory are severely restricted as they could potentially interact with the called function. To restrict the possibly accessed locations, low-level languages provide function and parameter annotations. The function level is discussed here and parameters in Section 3.5. In C/C++, functions can be marked as *pure* and *constant* via `__attribute__((pure/const))`. The `pure` annotation guarantees that the function will at most *read* global variables and not *access* any other location. The `const` annotation also disallows global reads. In LLVM-IR, similar annotations exist. A function can be marked as `readnone`, to indicate that no

1. `speculatable` (and `readnone`[1], ref. Section 3.4.1)
2. `readnone`
3. `readonly` and `inaccessiblememonly`
4. `readonly` and `argmemonly`
5. `readonly` and `inaccessiblemem_or_argmemonly`
6. `readonly`
7. `writeonly` and `inaccessiblememonly`
8. `writeonly` and `argmemonly`
9. `writeonly` and `inaccessiblemem_or_argmemonly`
10. `writeonly`
11. `inaccessiblememonly`
12. `argmemonly`
13. `inaccessiblemem_or_argmemonly`

Fig. 4: Optimistic function side-effect choices.

[1] The `speculatable` annotation is fairly new so we add the implied `readnone` explicitly.

memory is accessed, as `readonly` if there is no memory write, or as `writeonly` if there is no memory read. In addition, LLVM uses `inaccessiblememonly` to indicate that all accessed locations are not directly accessible from the user code, `argmemonly` to indicate that all memory accesses are based on pointer arguments, and `inaccessiblemem_or_argmemonly` to combine the two[2]. To exploit actual, not potential, behaviors, we generate optimistic opportunities with the optimistic choices listed in Figure 4. During the search space exploration (ref. Section 4.2), the choices are tried in order, thereby gradually decreasing the optimism.

3.4.3 Runtime Exceptions A function invocation can return to its respective call site, not terminate at all, or it can return to a point higher up the call chain. The latter, referred to as stack unwinding, is most often associated with runtime exceptions. Thus, if the called function raises an exception which is not caught inside that function invocation, the exception will traverse the call chain until a suitable handler is found. Since the code succeeding the in-between invocations would then be skipped, the compiler has to ensure the integrity of the program state prior to a potentially unwinding call. Hence, all non-local memory effects preceding an invocation that might transitively raise an exception have to be visible, and the side-effects after the invocation shall not be visible. As this severely limits the code movement and combination abilities only to preserve the semantics in case an exception is actually raised, it offers a perfect optimistic optimization opportunity for all programs, and program runs, that will not raise exceptions.

Compilers often allow to disable exceptions through options, e.g., `-fno-exceptions`. Additionally, C++ has the keyword `noexcept`, and the `nothrow` attribute is often supported. However, runtime exceptions are not the only cause for stack unwinding. We therefore use the LLVM-IR `nounwind` function attribute to guarantee each call site will either return control to its successor instruction, or not at all.

3.4.4 Return Values In addition to the side-effects, functions return values. While speculation on values opens up a far too large search space, there are common idioms that we optimize for. In particular, functions that return a value with the same type as one or multiple of their parameters might always return one of them.

To limit the number of optimistic opportunities, we only consider functions that return a pointer type. The number of optimistic choices is then equal to the number of parameters with the same type. The LLVM-IR parameter attribute `returned` is used to indicate that the return value is equal to the argument passed for this parameter. During the search space exploration (ref. Section 4.2), the suitable parameters are tried from the first one declared to the last. This is preferable because class methods take an implicit "`this`" object pointer, which is often returned.

3.4.5 Visibility To write modular and maintainable programs, most programming languages allow to choose different scopes for a symbol declaration. In particular, functions can be, among others, declared with a global or local scope. In C/C++, the former is the default while the latter, i.e., translation unit local, requires the function to be declared as `static`. Only if that is the case, the compiler can reason about *all* call sites prior to link time[3]. This can then justify more aggressive inlining as well as inter-procedural information propagation from call sites to the function definition[4].

[2] GCC's attribute `leaf` is similar to `inaccessiblemem_or_argmemonly` in LLVM-IR.

[3] Link time optimizations [15,12] are discussed in more detail in Section 5 and Section 6.

[4] While not in LLVM, a prototype for such a pass has been proposed already [8].

To limit the visibility, or scope, of a function declaration optimistically, we change the linkage type of external functions to internal. This is valid if, at link time, there are no users outside the current translation unit. If there are, the linking process, and thereby the verification, will automatically fail. Changing the linkage type of a function declaration in LLVM-IR to internal has a similar effect as the `static` keyword in C/C++.

3.5 Pointer Attributes

Pointers and the associated memory accesses, are arguably the most complicated part of a program. Especially in low-level languages, such as a compiler's intermediate representation, there are various caveats that have to be considered. Two memory accesses can for example alias, hence they might access (partially) the same memory locations. An access can be invalid at runtime if the accessed location is not dereferenceable, e.g., if the access pointer is "dangling". Similarly, the access can be invalid if the alignment of the access pointer violates the requirements of the assembly instruction that was chosen to implement it. As a consequence, potentially aliasing accesses induce dependences that have to be preserve similar to the control conditions of potentially invalid accesses.

3.5.1 Aliasing Since the use of unrestricted pointers is a major source of uncertainty during program optimization, compilers employ various forms of context-, flow-, type- and field-sensitive alias analyses [23,22,17,10,14,7]. Alias analyses, as well as the dependence analyses built on top, are tasked to identify and classify the dependence between side-effects. Only due to this information, transformations can decide if it is sound to alter the execution order of accesses, substitute them with already available values, or eliminate them all together. However, identifying aliasing pointers is on its own an undecidable problem [20]. Even if it is decidable for a given program, it is complex and consequently unrealistic to expect pointer related uncertainties to be resolved through static analyses alone [3].

Programming languages for which pointers by default alias commonly provide annotations to *restrict* the set of objects a pointer can alias with. While these annotations, e.g., `restrict`/`__restrict__` in C/C++, and `noalias` in LLVM-IR, are coarse-grained tools, they already allow to handle a common case: *Two pointers that do not originate in the same "restrict" qualified declarations cannot alias.*

We introduce the `noalias` annotation to function parameters and return values with pointer type. As the support for otherwise scoped restrict qualified pointers in LLVM is preliminary, we did not investigate this possibility for now.

3.5.2 Capturing Compilers try to determine the provenance, or the source object, of a pointer to rule out aliasing. Aliasing is impossible if a pointer is based on an object another pointer cannot be based on. An example are stack allocated objects that, *initially*, cannot alias with any pointer loaded from memory or provided from the outside. However, as soon as a pointer to the stack object *escapes*, i.e., the address of the object is potentially duplicated and made available to the rest of the program, this guarantee is void. A pointer conservatively escapes if it is passed to a function or stored in memory.

We augment the results of the already performed inter-procedural capture analyses in LLVM, which derives `nocapture` function parameter annotations, with optimistic annotations if they were not derived. For C/C++, Clang allows the programmer to achieve the same effect through `__attribute__((noescape))`.

3.5.3 Usage As a fine-grained supplement to the function side-effects described in Section 3.4.2, LLVM allows to annotate pointer parameters with access information. The choices again include `readnone`, to express that the pointer is not dereferenced during the execution of the function, `readonly`, to guarantee the absence of stores through the pointer, and `writeonly`, which rules out read accesses to the pointer.

The optimistic opportunity generated for each pointer parameter includes all three optimistic alternatives and is, again as the function side-effect equivalent, explored from the most optimistic one to the least. As before, if no optimistic choice could be successfully verified a pessimistic choice is taken, thus the pointer is not annotated.

3.5.4 Alignment There are different ways pointer alignment is exploited by a compiler. A very important one is the ability to utilize specialized instructions on machines that distinguish between aligned and unaligned memory accesses. Especially for vector code (SIMD) this can cause a significant performance difference.

For C/C++, compilers offer various ways to add alignment information including `__attribute__((aligned(N)))` qualifier, and the `__builtin_assume_aligned(P, N)` call. In this work, we introduced three different alignment annotations into the LLVM-IR. First, for memory accesses to describe their individual alignment, then for pointer parameters, and finally for pointers loaded from memory. In each case we provided two optimistic choices, cache line alignment and pointer alignment.

3.5.5 Dereferenceability Pointers might or might not point to a memory address that can be accessed at a certain program point. If they do not when accessed, the behavior is undefined. Consequently, compilers have to be especially careful when they move memory accesses which can easily prevent powerful optimizations such as loop hoisting or argument promotion.

As pointers most often point to memory that is in fact accessible, we can optimistically introduce the corresponding LLVM-IR annotation `dereferenceable(N_Bytes)`. It is used for function parameters and return values with pointer type, as well as to annotate pointers loaded from memory. In all three situations we have two optimistic choices, dereferenceability of a single element, or, alternatively, 64 consecutive elements. To achieve a similar effect for returned pointers in C/C++, i.e., to guarantee a certain number of accessible bytes *if the returned pointer is non-null*, GCC and Clang provide the `__attribute__((alloc_size(...)))` function annotation.

3.5.6 Memory Invariance The `const` keyword in C/C++ can be circumvented by a `const_cast` except for uses in certain variable declarations. Even though LLVM does not generally retain `const` information, it allows to annotate accesses as *invariant* which states that all executions will result in the same value.

To improve optimizations of memory loads, we use the LLVM-IR `invariant.load` annotation optimistically. It can act as an alternative to fine-grained alias annotations and as such enable load coalescing and load hoisting out of loops.

3.6 Overlapping and Inconsistent Annotations

The various annotations we introduce are not disjoint. In fact, it is possible that the optimistically annotated program contains logical inconsistencies. As an example take a function which we optimistically declared as *constant* (ref. Section 3.4.2), thus which can be assumed to be completely free of memory side-effects. While this annotation already provides a tight guarantee on the overall side-effects the function shall induce, our algorithm might still not be able to annotate all pointer parameter of this function as “read-only” or “not-accessed” (ref. Section 3.5.3). While such inconsistencies can potentially violate implicit preconditions of the optimization pipeline, they might also allow to enable optimistic transformations that would otherwise not have been possible. This is partially due to the granularity of the annotations and partially due to the multitude of ways analysis and optimization passes can query information.

4 Implementation Details

Our implementation[5] is split into three components. The first, thought to be provided by the application developer, is a benchmark description. It consists of benchmark specific information, for example the compilation flags, and instructions to verify the result, e.g., the invocation of the test suite. Additionally, the source files, or individual functions, chosen for optimistic optimization are identified. The second component is a transformation pass in the LLVM compiler. It is run at 14 locations in the otherwise original `-O2`/`-O3` pipeline. Every time it will identify optimistic annotation opportunities and, depending on the command line flags provided, either ignore them, act on them, or report them to the outside. The brains of our approach is located in a dedicated and external driver script. It will interpret benchmark description files, request optimistic opportunities from the compiler pass, and explore the space of optimistic choices until a timeout is reached or all opportunities have been resolved. Since early decision can impact the code and thereby change the opportunities available at a later point in the pipeline, it is important to perform the exploration iteratively, one annotation insertion point at a time. Not all opportunities described in Section 3 are exploited at every location. Instead, easily droppable annotations, e.g., for parallel loops (ref. Section 3.2), are placed only before they are used, e.g., prior to the loop vectorizer. Invariable annotations, e.g., for functions visibility (ref. Section 3.4.5), are introduced only once in the very beginning.

4.1 Granularity of Optimistic Opportunities

Optimistic information can often be added in different, potentially nested, granularities. As an example we can annotate a function declaration as a whole, all pointer arguments individually, or, as implemented, do both. While we choose a fine granularity for declarations, we did not yet investigate annotations on individual call sites. Depending on the compiler, finer-grained annotations, i.e., parameter vs. function annotation, and call site vs. declaration annotation, can improve the result. However, they can also easily cause overlapping and inconsistent annotations (ref. Section 3.6), increase tuning time, and lead to results that are harder to replicate through source code annotations.

To limit tuning time we eliminated opportunities early on. This means, (1) we do not add annotations if *any* of the possible optimistic choices is already present in the code, and (2) we accumulate opportunities into a single pick based on the kind and name of the value involved. Hence, every time an opportunity arises for a variable, we check if we can reuse the choice made earlier for the same opportunity kind and variable name. For example, all function parameters with the same name in a single translation unit are annotated the same. While this is especially useful for the control flow speculation explained in Section 3.3, it generally reduces the number of opportunities we explore.

[5] Please see `https://github.com/jdoerfert/PETOSPA` for the code and benchmarks.

4.2 Search Space Exploration

The space of potential choices for optimistic optimization opportunities is often too large to be searched exhaustively. This is partially because the order in which opportunities are resolved is important, e.g., earlier choices may interact with new ones, and because different optimistic opportunities are non-binary choices, e.g., the function side-effects explained in Section 3.4.2. Consequently, a globally optimal solution, measured for example by the number of optimistically resolved opportunities or the final performance, is unrealistic for any real program. Instead, we find a locally optimal solution where opportunity kinds are explored in a fixed order. This order is empirically chosen to allow our exploration algorithm to optimistically resolve many opportunities at once. When the verification failed, the number of optimistically resolved opportunities is split in half. If an opportunity is already tested in isolation, the optimism of the choice is decreased. After a less optimistic choice was fixed, we increase the number of tested opportunities again to potentially allow many choices at once.

5 Evaluation

We evaluated our approach on six proxy applications for high-performance codes described in Figure 5. While these codes are simplified, they retain much of the original complexity, making them authentic benchmarks for our approach. They especially already contain manual annotations, though, they are, as any production code would be, too complicated to provide *all valid annotations* manually. Several of the codes have few important kernels which encompass the vast majority of the runtime. Others have a long flat profile which is similarly common in practise. We also have variety within our annotated sections with large and small kernels, along with stand alone kernels and kernels with deep call paths. Beyond the code details, the benchmarks exhibit a variety of run time profiles, providing a range from compute to memory bound proxy applications.

The experiments were performed on an Intel(R) Xeon(R) CPU E5-2699 v3 (Haswell), running at 2.30GHz with 72 threads and 36 cores across two sockets. For each generated executable we collected 20 timings for a medium problem definition. The following discussion is based on the results shown in Figure 7.

5.1 RSBench (A)

RSBench simulates resonance representation cross sections lookups for nuclear reactor core Monte Carlo particle transport. It is a compute bound alternative to the XSBench kernel (ref. Section 5.2), the algorithm that is currently in use. RSBench heavily relies on the standard math library. As shown in Figure 7, we compiled RSBench 99 times during the tuning. It took 497 seconds to finish with all 240 optimistic opportunities and we achieved 20.6% speedup compared to the original.

During the tuning, we see two significant speedups, each $\approx 10\%$ compared to the baseline, both while working on the earliest of the 14 annotation points. The first improvement happened after alias (Section 3.5.1), wrapping (Section 3.1), exception (Section 3.4.3), visibility (Section 3.4.5), dereferenceability (Section 3.5.5), and alignment (Section 3.5.4) annotations were added in a single step. The second one while annotating function side-effects (Section 3.4.2), the last annotation kind at each insertion point.

For this compute heavy code the first significant speedup is visible after 15 compilations (of 99) which together took 98 seconds (of 497 seconds) to explore.

					Compilations		
Benchmark	**ID**	**Description**	**# Threads**	**Base Time**	**All**	**Succ.**	**New Vers.**
RSBench	(A)	Multipole resonance representation cross section lookup	72	8.56s	99	32	9 (28.1%)
XSBench	(B)	Macroscopic cross section lookups	1	75.13s	96	47	5 (10.6%)
PathFinder	(C)	Searches for 'signatures' within graph	1	363.50s	257	62	22 (35.5%)
CoMD	(D)	Classical molecular dynamics algorithms	72	44.70s	129	49	13 (26.5%)
Pennant	(E)	Unstructured mesh with radiation-hydro physics	1	33.66s	530	69	12 (17.4%)
MiniGMG	(F)	Geometric multigrid solver	1	6.10s	16	16	4 (25.0%)

Fig. 5: Benchmark name, identifier, and description are shown first, followed by the number of threads executing the optimized hotspots and the baseline execution time. Column six describe how often the benchmark was compiled during tuning, column seven shows how often the result was successfully verified, and the last column specifies how many of these verified versions were not bit-wise identical to the last one created before.

Sec.	3.1	3.1	3.1	3.2	3.3	3.4.2	3.4.3	3.4.4	3.4.5	3.5.1	3.5.2	3.5.3	3.5.4	3.5.5	3.5.1	3.5.5	3.5.4	3.5.4	3.5.5	3.5.6	Total
Det.	`nsw`	`nuw`	`gep`			function behavior				pointer argument					ret. ptr		mem	ptr	mem	load	
(A)	0	12	0	3	1	5/7	2	0	4	19	0	11	11	19	0	0	63/64	20/21	21	34/45	225/240
(B)	0	16/22	0	3	1	4	0	0	1/3	1/2	0	1/2	11	11	0	0	33/34	9/10	10	28	129/141
(C)	0	0	0	4	8/27	15/23	14	0	2	16	15	12	16	16	4	4	29/30	37/41	38	34/37	264/299
(D)	2	16	0	6	0/2	3/4	1	0	0	2	0	2	1/2	2	0	0	61/71	25/26	26	32	179/194
(E)	0	18/19	0	0	0	18/37	9/14	8	33/37	66/78	71	77/79	53/85	46	10	2	91/92	37	36/37	35/37	610/689
(F)	47	132	9	18	3	3	1	0	2	5	0	4	5	5	0	0	132	44	44	25	479/479

Fig. 6: Annotation opportunities identified and successfully exploited for the tested benchmarks (ref. Figure 5). The numbers denote how often optimistic choices were used for opportunities in the final program version (first value), as well as the total number of opportunities identified (second value). A single number is shown if both would be equal.

5.2 XSBench (B)

XSBench simulates the macroscopic cross section lookups that are the primary performance concern for nuclear reactor core Monte Carlo particle transport simulations. It is a memory intensive, semi random memory access code. Our evaluation focused on a serial run of the XSBench proxy application as the code is memory latency bound and the limitation of our memory system hides any performance changes in parallel runs. After 96 compilations, 422 seconds, and 141 optimistically annotated opportunities, the final executable shows a 15.6% speedup over the baseline.

The first optimistically annotated version performed even ≈ 18.13% better than the baseline. It contained 23 optimistic choices for alias and wrapping opportunities. The next three versions internalized functions and forfeited the speedup. It is not until 54 annotations later that we regain most of the performance gains. These 54 choices are spread over dereferenceability, alignment, and control flow (Section 3.3) annotations.

For this memory latency sensitive code, we find our best version in the middle of our optimistic annotation tuning after only 28 compilation (of 96) and 88 seconds. XSBench has many successful compilations that make no change in the resulting binary (marked as ○), especially in the second half of the tuning. This is interesting as evidence of the compilers inability to utilize the additional information.

5.3 PathFinder (C)

PathFinder is a memory latency sensitive graph traversal and search. We see a 17.3% speedup with 299 annotations after 257 compilations taking a total of 4259 seconds.

PathFinder is the code that has the most “new” versions (shown as ♦), i.e., successfully verified binaries that differ from the last. In total, 35% of all successful builds are (new) versions. Over all versions, a relatively steady performance increase is visible. There are two smaller drops that happen, and recover, while annotating a single opportunity kind, first memory invariance (Section 3.5.6), and then function side-effects. For PathFinder we make the least optimistic choices, totaling 11.7% of all opportunities, but additional information is consistently changing the executable.

After 96 compilations, taking 1194 out of the 4259 total seconds, the maximum speedup was almost reached. While the most significant improvement happens for an early insertion point, gains are made throughout the entire tuning.

5.4 CoMD (D)

CoMD is a molecular dynamics code which uses the Lennard-Jones potential. It is another compute heavy proxy application and shows a 4.6% speedup. Tuning introduces 194 annotations in 2614 seconds and spread over 129 compilations.

While the final result is faster than the baseline, we see slowdowns for intermediate versions. The first happens after annotating alias, wrapping, exception, dereferenceability, and alignment opportunities. The next version, still working on alignment, abruptly regains the loss. Later we experience a similar drop below the base line, again after annotating alignment and wrapping information. The majority of the optimistic opportunities are concerned with memory operations in this compute intensive code.

The final and best version is 4.62% faster than the baseline, but a speedup of 3.92% is already achieved after 21 compilations and 404 out of 2614 seconds.

5.5 Pennant (E)

Pennant is an unstructured mesh physics application using radiation–hydro code. Pennant's runtime has a long tail of small functions which limits (due to time) our ability to annotate more of the application. Our tuning is unable to make any performance gain despite adding 689 annotations over the course of 530 compilations.

While no speedup was achieved, we discovered an intermediate version with a significant slowdown. This version has only five additional optimistic annotations compared to the one before. The slowdown, as well as the subsequent recover, happens while we annotate function memory effects, an opportunity with 13 different optimistic choices. The five annotations which cause the slowdown, along with the five that recover it again, are annotated through 119 compilations. Thus, our search algorithm was forced to reduce the optimism of the individual choices until verification succeeded.

The Pennant code is unable to capitalize on the additional information despite 610 optimistic choices made for 689 opportunities. During most of the tuning (observe the logarithmic axis) additional annotations did not change the binary.

5.6 MiniGMG (F)

MiniGMG is a benchmark for geometric multigrid solvers. It is designed to stress both the compute and memory subsystem of the hardware. MiniGMG has shown no performance changes after annotating all 479 opportunities optimistically.

MiniGMG has the most regular results. Each of the four versions was followed by three successful compilations, which did not change the binary. None of the version showed any significant change in performance. The two opportunity kinds wrapping and alignment account for over half of all opportunities.

5.7 Successfully Verified Annotations

The dots ○ in Figure 7 indicate successfully verified builds that contain more annotations but do not change the resulting binary. Depending on the benchmark, between 10.6% and 35.5% of valid builds resulted in a new binary which we had to verify. The other cases were versions bit-equal to the last which were not verified. XSBench (B) only produces five different versions despite 47 successful builds with new annotations. In contrast, PathFinder (C) creates 22 different binaries in 62 successful builds. XSBench and Pennant (E) both have substantial successful compilations after the final version is first compiled. PathFinder is the only benchmark that continues to make improvements late, however these are only regaining lost performance from earlier optimistic versions.

5.8 Optimistic Choices

Over all benchmarks, a large percent (>88%) of opportunities result in optimistic choices (see Total in Figure 6). This holds to the understanding that there is a great deal of information that the compiler is not aware of. The hope is to help the developer understand what information will most likely generate a positive effect on the application. At the same time we need to remedy limitations in current compilers to make profitable use of additional knowledge. The annotation pass run first in the optimization pipeline discovers the majority of the optimistic opportunities (always >70%). This is not surprising as optimistic information is often maintained throughout the pipeline. As a consequence, we will, for example, explore function interface specific opportunities only at the first (of the 14) insertion points.

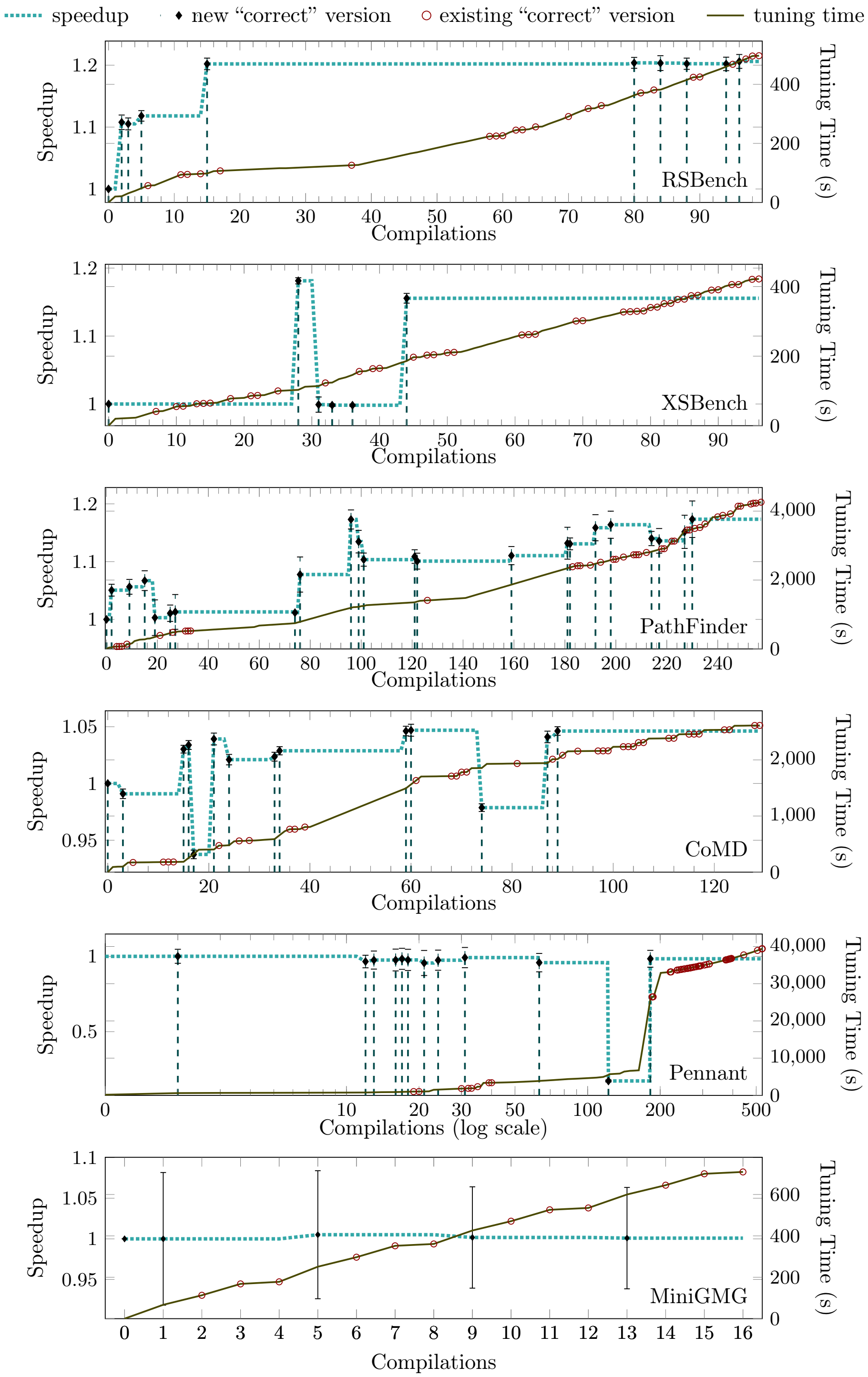

Fig. 7: Performance and optimization results for the six high-performance proxy applications described in Figure 5. Each plot shows the speedup (left) relative to the original version, and the tuning time (right), both with regards to the number of compilations (=tries) performed. If annotations yielded a successfully verified executable we mark it as ♦, if it was different from the last one, or as ○, if it was not.

5.9 Comparison with Link Time Optimization (LTO)

We also collected data for both (monolithic) LTO and thin-LTO [15]. Figure 8 show the performance gap determined by our technique with LTO/thin-LTO as a baseline. While the difference is smaller than for non-LTO builds, it remains significant, i.a., our optimistically annotated XSBench shows a 14% speedup compared to a full monolithic LTO build.

Compile time over the original source increased through monolithic LTO by 5.5% to 18.5%. With (sequential) thin-LTO the increase was between 3.6% and 17.3% (expect for MiniGMG (F) which showed a compile decrease of 15.6%). For the optimistically annotated benchmarks compile time decreased by 0.3% to 2.5%.

Proxy	**LTO**	**thin-LTO**
(A)	2.86%	5.68%
(B)	14.03%	41.23%
(C)	3.67%	4.79%
(D)	4.75%	4.48%
(E)	-1.13%	-1.14%
(F)	0.73%	0.79%

Fig. 8: Performance compared to monolithic and thin-LTO.

6 Related Work

While we are not aware of exiting work that makes similar use of additional optimistic static information to identify performance gaps through the optimizations in an exiting compiler, there are various related research fields.

Autotuning Given our moderate knowledge in the area of autotuning we restricted ourselves to the most important files and functions in the evaluated benchmarks. Consequently, it was sufficient to use ad-hoc search space reductions and a custom search space exploration to determine optimistic choices. To allow the approach to scale in the future we need to incorporate elaborate tuning mechanism as offered through tools like OpenTuner [4] or BOAT [6]. The latter seems explicitly interesting as we can integrate domain knowledge, e.g., we could leverage information such as the expected benefit based on the opportunity kind.

Link Time Optimization Certain inter-procedural uncertainties are already resolvable through link time optimization (LTO). While existing LTO implementations in GCC [12] and LLVM [15] have shown great success, it is unrealistic to assume they will ever reach the same level of inter-procedural information that can be provided through optimistic annotations. There are two main reason. First, only where LTO compilation was used, link time inter-procedural information can be collected. Thus, system or third party library calls will often limit the analyses results as external functions call do it in non-LTO compilation. Second, LTO enabled compilation suffers still from input and context dependent uncertainties. Even if we assume we could inline all function calls or derive perfect caller-callee information statically, nine of the 20 optimistic opportunities we collected would still be needed. Finally, LTO approaches induce a constant compile time penalty as discussed in Section 5.9.

Super Optimization Our technique shares ideas and goals with super optimization approaches [25,21] as well as other aggressive optimization techniques [24,18] developed outside of a classic compilation toolchain. While these techniques are often focused on correctness first, e.g., through semantic encodings or rewrite systems, and performance second, we relaxed the correctness criterion and put the user in charge of verification. We also do not introduce or explore new transformations but instead try to enable existing ones. An interesting future direction is the combination of the reasoning capabilities common to super optimizations with an optimistic approach to identify the most promising opportunities. Even if complete static verification might be out of reach, runtime check based verification has shown great success in the LLVM loop vectorizer and polyhedral optimizer Polly [3,9].

7 Conclusion & Future Work

Our findings show that there is extensive knowledge, which may be apparent to the developer, that the compiler is unable to discover statically. This information, once exposed to the compiler, can significantly improve performance. However, additional information will most often not result in better performance or even a different executable, either because it is unusable or unneeded for optimizations, suitable optimizations are simply missing, or later analyses would have determined it as well.

Beyond the integration of new opportunities, we plan to isolate interesting optimistic choices automatically. Those with the most significant performance impact, the ones without any impact at all, as well as those causing a regression, may all provide valuable information. Optimally, we want to predict what informational will be used, and what annotations are necessary to achieve a performance gain. In addition, we want to hone in on annotations producing a performance loss because these indicate compiler flaws.

8 Acknowledgments

We would like to thank the reviewers for their extensive and helpful comments.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

References

1. Ahmad, D.: The Rising Threat of Vulnerabilities Due to Integer Errors. IEEE Security & Privacy **1**(4) (2003), `https://doi.org/10.1109/MSECP.2003.1219077`
2. Allen, J., Kennedy, K., Porterfield, C., Warren, J.: Conversion of Control Dependence to Data Dependence. In: ACM Symposium on Principles of Programming Languages, Austin, Texas, USA (1983), `https://doi.org/10.1145/567067.567085`
3. Alves, P., Gruber, F., Doerfert, J., Lamprineas, A., Grosser, T., Rastello, F., Pereira, F.M.Q.: Runtime Pointer Disambiguation. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA (2015), `http://doi.acm.org/10.1145/2814270.2814285`
4. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U., Amarasinghe, S.P.: OpenTuner: An Extensible Framework for Program Autotuning. In: International Conference on Parallel Architectures and Compilation, PACT. ACM (2014), `https://doi.org/10.1145/2628071.2628092`
5. Collard, J., Barthou, D., Feautrier, P.: Fuzzy Array Dataflow Analysis. In: ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP) (1995), `http://doi.acm.org/10.1145/209936.209947`
6. Dalibard, V., Schaarschmidt, M., Yoneki, E.: BOAT: Building Auto-Tuners with Structured Bayesian Optimization. In: International Conference on World Wide Web, WWW. ACM (2017), `https://doi.org/10.1145/3038912.3052662`

7. Diwan, A., McKinley, K.S., Moss, J.E.B.: Type-Based Alias Analysis. In: Conference on Programming Language Design and Implementation (PLDI) (1998), http://doi.acm.org/10.1145/277650.277670
8. Doerfert, J., Finkel, H.: Compiler Optimizations for OpenMP. In: International Workshop on OpenMP (IWOMP) (2018), https://doi.org/10.1007/978-3-319-98521-3_8
9. Doerfert, J., Grosser, T., Hack, S.: Optimistic Loop Optimization. In: International Symposium on Code Generation and Optimization, CGO (2017), http://dl.acm.org/citation.cfm?id=3049864
10. Emami, M., Ghiya, R., Hendren, L.J.: Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In: Conference on Programming Language Design and Implementation (PLDI) (1994), http://doi.acm.org/10.1145/178243.178264
11. Feautrier, P.: Dataflow analysis of array and scalar references. International Journal of Parallel Programming (1991), https://doi.org/10.1007/BF01407931
12. Glek, T., Hubicka, J.: Optimizing real world applications with GCC Link Time Optimization. CoRR (2010), http://arxiv.org/abs/1010.2196
13. Grosser, T., Größlinger, A., Lengauer, C.: Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. Parallel Processing Letters (2012), https://doi.org/10.1142/S0129626412500107
14. Jeong, S., Jeon, M., Cha, S.D., Oh, H.: Data-driven context-sensitivity for points-to analysis. PACMPL (2017), http://doi.acm.org/10.1145/3133924
15. Johnson, T., Amini, M., Li, D.X.: ThinLTO: scalable and incremental LTO. In: International Symposium on Code Generation and Optimization, CGO (2017), http://dl.acm.org/citation.cfm?id=3049845
16. Lattner, C., Adve, V.S.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: International Symposium on Code Generation and Optimization CGO (2004), https://doi.org/10.1109/CGO.2004.1281665
17. Lattner, C., Lenharth, A., Adve, V.S.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Conference on Programming Language Design and Implementation (PLDI) (2007), http://doi.acm.org/10.1145/1250734.1250766
18. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with alive. In: Conference on Programming Language Design and Implementation (PLDI) (2015), https://doi.org/10.1145/2737924.2737965
19. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. In: Conference on Supercomputing (SC) (1991), http://doi.acm.org/10.1145/125826.125848
20. Ramalingam, G.: The Undecidability of Aliasing. Trans. Program. Lang. Syst. (1994), http://doi.acm.org/10.1145/186025.186041
21. Sasnauskas, R., Chen, Y., Collingbourne, P., Ketema, J., Taneja, J., Regehr, J.: Souper: A Synthesizing Superoptimizer. CoRR (2017), http://arxiv.org/abs/1711.04422
22. Shapiro, M., Horwitz, S.: Fast and Accurate Flow-Insensitive Points-To Analysis. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (1997), http://doi.acm.org/10.1145/263699.263703
23. Steensgaard, B.: Points-to Analysis in Almost Linear Time. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (1996), http://doi.acm.org/10.1145/237721.237727
24. Tate, R., Stepp, M., Lerner, S.: Generating compiler optimizations from proofs. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 389–402. ACM (2010), https://doi.org/10.1145/1706299.1706345
25. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2009), https://doi.org/10.1145/1480881.1480915