A Collaborative Dependence Analysis Framework

Nick P. Johnson [*] Jordan Fix
Stephen R. Beard
Taewook Oh [†]
Princeton University, USA
{npjohnso,jfix,sbeard,twoh}@cs.princeton.edu

Thomas B. Jablin
UIUC / Multicoreware, USA
jablin@illinois.edu

David I. August
Princeton University, USA
august@princeton.edu

Abstract

Compiler optimizations discover facts about program behavior by querying static analysis. However, developing or extending precise analysis is difficult. Some prior works implement analysis with a single algorithm, but the algorithm becomes more complex as it is extended for greater precision. Other works achieve modularity by implementing several simple algorithms and trivially composing them to report the best result from among them. Such a modular approach has limited precision because it employs only one algorithm in response to one query, without synergy between algorithms. This paper presents a framework for dependence analysis algorithms to collaborate and achieve precision greater than the trivial combination of those algorithms. With this framework, developers can achieve the high precision of complex analysis algorithms through collaboration of simple and orthogonal algorithms, without sacrificing the ease of implementation of the modular approach. Results demonstrate that collaboration of simple analyses enables advanced compiler optimizations.

Keywords Collaborative analysis; Demand-driven analysis; Dependence analysis; Program Dependence Graph.

1. Introduction

Program dependence analysis is critical to a broad range of static analysis and program transformation techniques. These include compiler optimizations (such as instructions scheduling, loop invariant code motion, polyhedral techniques, vectorization, and parallelization), bug finding tools, program slicing, and many other techniques. Consequently, decades of research have uncovered a broad array of analysis algorithms to increase the accuracy and precision of dependence analysis. Pointer analysis (including points-to analysis and alias analysis) judges whether two pointers within a program reference the same memory location [1, 3, 5, 18, 21, 32, 33]. Shape analysis (including heap reachability analysis) models the connectivity of objects to answer questions of disjointness or cyclicity of data structures as a whole [9, 10, 30]. Loop dependence analysis determines whether memory accesses from different iterations of a loop are ordered [2, 28]. Research continues since these problems are difficult and not solved; the general case is undecidable and various decidable abstractions are intractable [13, 26]. Each proposal is an approximation and occupies a distinct niche in the trade-off between precision and scalability [12], but no algorithm dominates all others.

One means to achieve a desired precision-scalability compromise is to extend algorithms with additional rules or more precise models [3, 5, 18, 21, 33]. However, this complicates algorithms, as each new rule must be considered in relation to existing rules.

Alternatively, one may combine several algorithms, thus blending their performance and accuracy [4, 7, 11, 16, 17, 20, 23, 27]. Some of these prior works hide different models as implementation details behind a common interface to enable composability among algorithms [17, 24]. These approaches design their member algorithms to act independently and in isolation. The composition reports the most precise result from any of its members; the composition is the sum of its parts.

Other works combining multiple algorithms additionally provide a property we call *collaboration* – when these multiple algorithms work synergistically together. Two algorithms collaborate if the combination disproves dependences which neither algorithm disproves in isolation. For instance, linear integer arithmetic (LIA) may disprove aliasing among affine array accesses, but cannot show that two pointers must reference the same array. Shape analysis complements LIA by reasoning about pointers [9, 10, 30]. Through collaboration, the pair uses both logics to service a single query.

Past works including this notion of collaboration either focus on abstract or specialized domains such as SMT solvers which do not readily lend themselves to dependence analysis [7, 16, 27], or combine some specific analyses together to demonstrate their synergy and achieve better precision or efficiency [3, 4, 20].

Instead, this proposal presents a framework and language through which analysis algorithms found in real world compilers optimizing complex general purpose programs can collaborate in the domain of dependence analysis. Developers using this framework can achieve the high precision of complex analysis algorithms through collaboration of simple and orthogonal algorithms, without sacrificing the ease of implementation of the modular approach.

[*] Work performed at Princeton University. Now at D.E. Shaw Research.

[†] Work performed at Princeton University. Now at Facebook.

In this framework, algorithms are connected in a chain through which a query is passed until an algorithm can definitively answer the query. The framework's interface provides a common language of queries to unite different algorithms. To resolve a query, an algorithm may need to prove a premise that is beyond its own logic. The algorithm formulates such premises as new queries, optimistically assuming that these queries are analyzable by another logic in the chain. The framework connects the premises to the beginning of the chain, forming an *ensemble* of analyses, to allow any algorithm to service any premise via implicit recursion. Such algorithms that break an analysis query into subproblems are referred to as *functor algorithms*.

Because ensembles support collaboration, developers may modularize the development of analysis algorithms through *factorization*. Instead of increasingly complicated algorithms that incorporate additional types of reasoning, factorization achieves precision through many simple algorithms. Each algorithm disproves queries within its core competence and assumes other algorithms provide the necessary diversity of logic to solve its premises. Factored algorithms are developed independently without requiring knowledge of others, letting developers easily extend analysis precision to the needs of a client. To date, we have had multiple authors contributing to our infrastructure write twenty-three separate algorithms, covering a wide range of different types of analyses such as pointer analyses, shape analyses, and reachability analyses. These algorithms were implemented independently to the needs of each author, yet have proven valuable to all users of the infrastructure.

The framework achieves great precision through collaboration while maintaining the composability of prior approaches. This paper presents the framework, including:

- a demand-driven interface for contextualized dependence queries, which serves as a common vocabulary to unify disparate analysis algorithms;
- premise queries to enable collaboration among many simple, modular analysis algorithms; and,
- the functor design pattern, enabling factored analysis algorithms which compose into ensembles.

This paper describes 13 analysis algorithms implemented in the framework (Section 4) and evaluates their precision and collaboration on the SPEC benchmarks (Section 5).

2. Background

Algorithms for dependence analysis answer non-trivial properties of a program. Such properties are undecidable, and thus each algorithm makes a best effort, reporting *definitely yes*, *definitely no*, or *unknown* when the answer is beyond the logic of the algorithm. These unknown cases are the vehicle for composition; given a diversity of analysis logics, some algorithms answer definitively when others cannot. In such cases, a composition of algorithms reports the definite answer. The composition is sound if its members are all sound [17].

Composability relies on a strict interface for each analysis algorithm. The interface provides a common language of queries to unite different algorithms. The language abstracts the disparate models employed in each algorithm yet is general enough that it does not hinder its member algorithms.

Collaboration describes algorithms which are more precise in concert than alone. Collaboration is stronger than composability since it supports factored algorithms. Factorization achieves precise results through independently developed, modular algorithms, thus simplifying development.

2.1 Dependence Analysis

The *memory dependence* relation is a primary motivation. Dependence has three conditions. We say there is a memory dependence from instruction i_1 to instruction i_2 iff (*alias*) the footprint of operation i_1 *may-alias* the footprint of i_2, and (*feasible-path*) there is a feasible path of execution from i_1 to i_2 which (*no-kill*) does not contain an operation which overwrites the common memory footprint. Footprint denotes the set of memory locations read or written by the instruction.

3. Design

We describe the design in a top-down fashion, first presenting the composition of analysis algorithms in Section 3.2, then the language of queries in Section 3.3. For clarity, we present each aspect of the design in the context of a sample algorithm which we call *Array of Structures* (AoS).

3.1 An Example Analysis Algorithm

Analysis algorithms for the dependence relation attempt to disprove one or more of the conditions of dependence (Section 2.1). For example, AoS disproves dependences between memory operations by disproving a certain class of aliasing between pointers to nested aggregate types. Consider a query to test a dependence between two operations. Suppose that the first operation accesses a pointer of the form A=`&a[`i_1`]...[`i_n`]` and that the second operation accesses a pointer with similar construction B=`&b[`j_1`]...[`j_n`]`. If $a = b$ and have the same declared type and if there exists a position k such that indices $i_k \neq j_k$, then the pointers cannot alias and AoS reports no dependence. If the pointers do not match that form, the algorithm is inapplicable. When applicable, this simple logic is powerful: it disproves aliasing, even if indices i_q, j_q at other positions $q \neq k$ are non-affine or otherwise incomparable. This algorithm ignores most cases; it is designed to serve as part of a diverse ensemble.

3.2 Ensembles Collaborate by *Topping Premises*

This section describes the combination of analysis algorithms into an ensemble, and how *topping* enables collaboration within an ensemble. We defer discussion of the query language until Section 3.3; For now, the reader need only understand that a *query* tests the relationship between its specified operations or pointers.

Viewed in isolation, an analysis algorithm receives incoming queries and returns a definitive result if the query is recognized by the algorithm's reasoning. The framework places no constraints on the internal operation of an algorithm, yet an expected work flow guides the design.

Our framework recognizes that every algorithm has limitations, yet analyzing one query may require reasoning that transcends one algorithm. In our example (Section 3.1), AoS

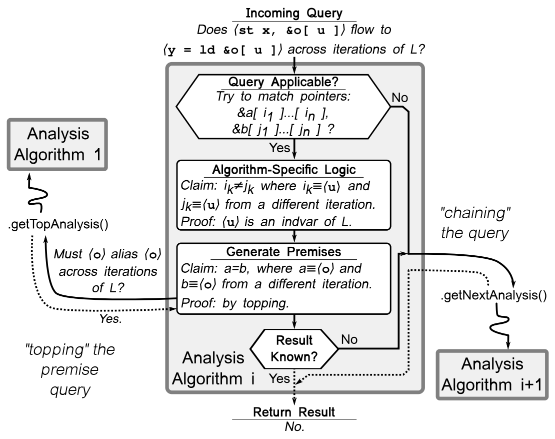

Incoming Query
Does ⟨st x, &o[u]⟩ flow to ⟨y = ld &o[u]⟩ across iterations of L?
Query Applicable?
Try to match pointers: &a[i1]...[in], &b[j1]...[jn] ?
No
Yes
Algorithm-Specific Logic
Claim: ik≠jk where ik≡⟨u⟩ and jk≡⟨u⟩ from a different iteration.
Proof: ⟨u⟩ is an indvar of L.
Generate Premises
Claim: a=b, where a≡⟨o⟩ and b≡⟨o⟩ from a different iteration.
Proof: by topping.
Result Known?
No
Yes
Analysis Algorithm i
Return Result
No.
Analysis Algorithm 1
.getTopAnalysis()
Must ⟨o⟩ alias ⟨o⟩ across iterations of L?
Yes.
"topping" the premise query
"chaining" the query
.getNextAnalysis()
Analysis Algorithm i+1

Figure 1: Flowchart depicting the internal operation of the AoS analysis algorithm (Section 3.1).

logic proves disjointness of array accesses, but does not contain any means to compare base pointers. Since the proposition $a = b$ is outside of its logic, AoS isolates it as a *premise*.

An algorithm extracts premises from incoming queries to isolate propositions which its own logic cannot solve. The algorithm formulates premises as new queries and *tops* them by sending them back to the top of the ensemble as a new query, signifying that solving premise queries will contribute to solving the original query. If the algorithm cannot determine a definitive answer or deems the query not applicable, then it *chains* the query by sending it to the next algorithm in the ensemble. These two possibilities of either topping or chaining are seen in Figure 1.

Suppose the example algorithm AoS (Section 3.1) receives a query and demonstrates that $i_k \neq j_k$. The second condition $a = b$ remains. One could introduce ad-hoc rules for comparing base pointers, but these are limited; instead, this section discusses how the ensemble allows AoS to prove $a = b$ by issuing a *premise* query to the ensemble.

3.2.1 Combining Analyses into an Ensemble

To form an ensemble, each algorithm is composed such that one algorithm's chained queries feed the next. The chain ends with a null algorithm which always reports the most conservative answer.

Queries enter the ensemble's top-most member. These queries may originate from either a client or from another analysis algorithm "topping" a premise it generates (see Figure 2). Because both client and premise queries use the same query language (described in Section 3.3), all analysis algorithms understand both query types; the query type is irrelevant to the analysis algorithms.

Without topping, composition by chaining resembles composition in LLVM [17]. Topping can significantly improve an ensemble's performance, as evaluated in Section 5.5. We compare to LLVM in Section 6.

Algorithms at higher positions in the ensemble may return a definitive answer before subsequent members receive the query. Algorithms which quickly recognize many queries should be placed above slower ones or those with lesser applicability, thereby filtering the easy queries. The developer

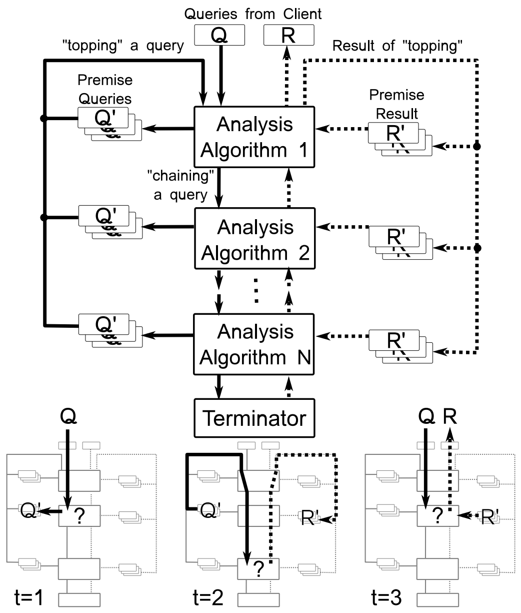


Figure 2: *(above)* An ensemble of N algorithms. Premise queries re-enter at the top; chained queries pass to the next algorithm. *(below)* Path of a query Q over time. $t = 1$: Algorithm 2 deconstructs Q into a premise query Q'. $t = 2$: Algorithm N solves Q' and returns R'. $t = 3$: Algorithm 2 receives R' and combines it with other information to yield the result R for Q.

assigns each algorithm a *scheduling priority* to control its relative position, lifting efficient algorithms above slow ones to reduce common-case query latency (Section 5.6).

Continuing with the example algorithm from Section 3.1, suppose that AoS is Analysis Algorithm 2 in Figure 2. To determine if $a = b$, AoS issues a premise query Q', which is topped to the beginning of the ensemble of analysis algorithms. Analysis Algorithm 1 will then examine Q' and determine if it is within its competency to resolve. If so it will produce a result R', else it will chain the query to the next analysis algorithm. As an example, assume that a and b are constant pointers and are initialized to point to the same global object. An analysis algorithm such as Unique Access Paths (UAP), described in Section 4.2.2, could determine that $a = b$. If UAP was Analysis Algorithm 1, it would return this result R' to AoS. With an answer to all of its premises, one of which was resolved via collaboration with UAP, AoS could respond to the original query it received.

3.2.2 Ensuring Termination

Topping a premise query introduces a cycle into the ensemble. Developers must ensure that cycles do not cause infinite recursion. Such assurances resemble familiar termination arguments: define a query complexity metric with a lower bound and demonstrate that premises have a lower metric than the original queries. Choice of metric depends on the particular analysis algorithm, but we demonstrate a few.

We prove termination in the AoS example with this metric: indexing operations (LLVM's `getelementptr` instruction) have a metric one greater than their base operand; all other operations have a metric of zero. AoS strips one layer

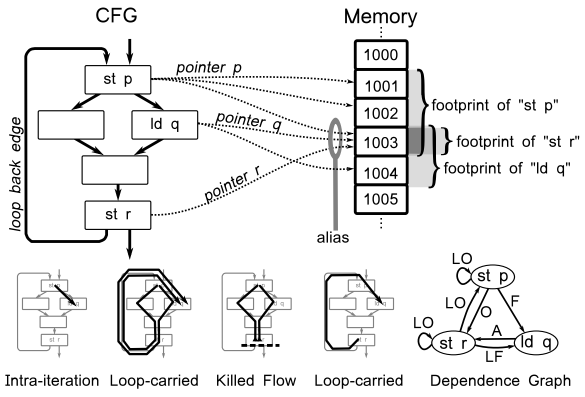


Figure 3: (*top*) Loads (`ld`) and stores (`st`) access memory via pointers. Pointers *alias* if they reference a common location. A *footprint* lists resources which an operation accesses. (*bottom*) Intra-iteration path from `st p` to `ld q`; Some loop-carried paths from `st p` to `ld q`; Operation `st r` kills loop-carried flows from `st p` to `ld q`; but, operation `st p` does not kill all loop-carried flows from `st r` to `ld q`; *(F)low*, *(A)nti*, and *(O)utput* dependences, *(L)oop-carried.*

of array indexing, thereby reducing the metric of the premise query. The metric's lower bound precludes infinite topping.

When several algorithms generate premise queries, the termination argument must consider all, either by using a single metric universally, or by demonstrating that no algorithm's premises increase another algorithm's termination metric. This kind of reasoning is easy in practice.

3.3 The Query Language

The query language enables the expression of dependence analysis queries between clients and analysis algorithms, and between the analysis algorithms themselves.

3.3.1 Specifying Shared Resources

Our proposed interface favors an implicit representation of resources, rather than explicit points-to sets, because it supports a wider variety of analysis algorithms. Our query semantics are defined around the *footprint* of a target operation. This footprint represents all resources (memory locations, or a special `IO` object to model side-effects) accessed by the target operation (see Figure 3). Algorithms may interpret footprints without enumerating their members and with varying degrees of precision. This loosens restrictions on an algorithm's internal operation and enables composability across disparate internal models.

In our example (Section 3.1), the target resource is the memory footprint of the target operation, i.e., the storage referenced by pointer B. The first operation's footprint may share those resources.

3.3.2 Loops and Temporal Relations Qualify Queries

Our proposal specifies context and paths via the *loop* and *temporal relation* parameters. The optional loop parameter scopes the query such that the static operations refer only to those dynamic instances which execute within an iteration of the loop. The temporal relation allows the client to query loop-carried and intra-iteration dependences separately.

The loop additionally demarcates control flow boundaries at loop iterations, and the temporal relation T selects paths with respect to those boundaries (see Figure 3). If T = Before, the first operation executes in a strictly-earlier iteration than the second; if T = After, the first operation executes in a strictly-later iteration than the second; and, if T = Same, both operations execute in the same iteration. If the loop is `null`, the query represents any dynamic instances of the operations along any feasible path.

The example algorithm (Section 3.1) attempts to prove that $i_k \neq j_k$. If both are defined as the same induction variable, the indices are equal when T = Same yet are unequal when T = Before or T = After (see Figure 1).

3.3.3 Types of Queries

The interface allows queries for three different relations: `modref_ii`, `modref_ip`, and `alias_pp`, summarized in Table 1. `modref_*` queries determine whether a operation i_1 modifies (`Mod`) or reads (`Ref`) some set of resources, returning `None`, `Mod`, `Ref`, or `Mod-Ref`. A *target* parameter implicitly specifies a set of target resources. For `modref_ii`, the target is the resource footprint of a target *operation* i_2, whereas for `modref_ip`, the target is the set of memory locations referenced by target *pointer* p_2 and *access size* s_2.

The mod-ref relation is similar to the may-depend relation. One determines if there is a memory dependence from operation i_1 to i_2 within a single iteration of loop L, by issuing two queries: `modref_ii`$(i_1,$ `Same`$, i_2, L)$ and `modref_ii`$(i_2,$ `Same`$, i_1, L)$. Similarly, one determines if there is a memory dependence from operation i_1 to i_2 across iterations of loop L, by issuing the queries: `modref_ii`$(i_1,$ `Before`$,$ $i_2, L)$ and `modref_ii`$(i_2,$ `After`$, i_1, L)$. The results of both queries determine if there is a flow dependence, output dependence. have more than one type of dependence because it both reads and writes memory, for instance, a call site. Most clients ignore read-after-read dependences.

4. Implementation

This section describes how we developed our analysis algorithms and briefly describes the operation of 13 of them.

4.1 Developing New Analysis Algorithms

While developing compiler transformations, we often observe the compiler acting conservatively due to analysis imprecision. Such imprecision indicates a deficiency of the ensemble and an untapped niche in the design space for a new algorithm. We describe a process to develop new algorithms which are precise in these cases. Developing algorithms in the proposed framework is no more complicated than developing a monolithic algorithm because composability allows the developer to remain largely ignorant of other algorithms in the ensemble. Collaboration frequently *simplifies* the process by addressing smaller algorithms in isolation.

The developer lists dependence queries with imprecise results, either manually or with a tool that compares static analysis to profiling. The developer confirms that each query is imprecise by arguing *why* a dependence cannot manifest. The developer generalizes this argument into an algorithm to

Semantics	Query	Context and Path Qualifiers
May instruction i_1 write to (`Mod`*) or read from (*`Ref`*) the resource footprint of target operation i_2?* Return `None`, `Mod`, `Ref`, or `Mod-Ref`.	`modref_ii`$(i_1,$ `Before`$, i_2, L)$	i_1 executes in iteration τ_1 of loop L and i_2 executes in some later iteration $\tau_2 > \tau_1$.
	`modref_ii`$(i_1,$ `Same`$, i_2, L)$	i_1 and i_2 both execute in the same iteration of L.
	`modref_ii`$(i_1,$ `After`$, i_2, L)$	i_1 executes in iteration τ_1 of loop L and i_2 executes in some earlier iteration $\tau_2 < \tau_1$.
May instruction i_1 write to (`Mod`*) or read from (*`Ref`*) the resource footprint of target pointer p_2 of length s_2?* Return `None`, `Mod`, `Ref`, or `Mod-Ref`.	`modref_ip`$(i_1,$ `Before`$, p_2, s_2, L)$	i_1 executes in iteration τ_1 of loop L and values of p_2 are computed in later iterations $\tau_2 > \tau_1$.
	`modref_ip`$(i_1,$ `Same`$, p_2, s_2, L)$	i_1 executes in the same iteration as p_2.
	`modref_ip`$(i_1,$ `After`$, p_2, s_2, L)$	i_1 executes in iteration τ_1 of loop L and p_2 computes pointers in earlier iterations $\tau_2 < \tau_1$.
May any memory location which is referenced by a dynamic pointer value computed by operation p_1 of length s_1 alias with the resources referenced by target pointer p_2 of length s_2? Return `No Alias`, `May Alias`, or `Must Alias`.	`alias_pp`$(p_1, s_1,$ `Before`$, p_2, s_2, L)$	p_1 is computed in iteration τ_1 of loop L and p_2 is computed in some later iteration $\tau_2 > \tau_1$.
	`alias_pp`$(p_1, s_1,$ `Same`$, p_2, s_2, L)$	p_1 and p_2 are computed during the same iteration of L.
	`alias_pp`$(p_1, s_1,$ `After`$, p_2, s_2, L)$	p_1 is computed in iteration τ_1 of loop L and p_2 is computed in some earlier iteration $\tau_2 < \tau_1$.

Table 1: Types of queries: `modref_ii` compares the footprint of the first instruction to the footprint of the target instruction; `modref_ip` compares instead to the resources referenced by a target pointer; `alias_pp` compares two pointers.

Analysis Algorithm	Sensitivity				Demand-driven?	Num. premise queries issued
	Memory-flow	Control-flow	Array/field	Calling-context		
Array of Structures	×	×	✓	×	Fully	1
Auto-restrict	×	×	×	✓	Partially	0
Basic Loop	×	×	✓	×	Fully	Many
Callsite	✓	✓	×	✓	Fully	Many
Global Malloc	✓	×	×	×	Partially	0
Kill Flow	✓	✓	×	×	Fully	Many
No-Capture Global	×	×	×	×	Fully	0
No-Capture Source	×	×	×	×	Fully	0
PHI Maze	×	×	×	×	Fully	0
Semi-Local	×	×	×	×	Partially	Many
Unique Paths	✓	×	✓	✓	Partially	Many
Disjoint Fields	×	×	✓	×	Partially	0
Field Malloc	×	×	✓	×	Partially	1

Table 2: Summary of the analysis algorithms described and evaluated.

recognize such queries and report independence. Algorithms discovered through this process are largely *orthogonal* (see Section 5.7) to the ensemble and target queries which affect the client and which occur in common programs.

The developer may delegate parts of its analysis logic to other implementations through premise queries. As a simplifying assumption, our design process encourages developers to think of topping as an oracle: the ensemble is strong and will only get stronger. One risk of this approach is that the ensemble may not be strong enough to disprove the premises. Another risk is increased query latency. While not an issue for most algorithms, this can be mitigated by lowering its scheduling priority (Section 3.2.1).

4.2 Analysis Implementations

We have implemented 23 analysis algorithms in this framework. For space, this paper describes and evaluates only 13 which we chose for their orthogonality (Section 5.7).

Section 2.1 defines three conditions for dependence: *may-alias*, *feasible-path*, and *no-kill*. Each algorithm attempts to disprove a dependence according to one or more of these conditions. All algorithms operate at the compiler IR level. Table 2 summarizes the algorithms evaluated in this paper. We describe each algorithm's behavior by describing types of dependences which they disprove; other queries chain to the next algorithm in the ensemble by default.

4.2.1 Base Analysis Algorithms

Several of our algorithms perform basic reasoning about various idioms which appear within the IR. These algorithms do not generate premise queries.

Basic Loop is a straightforward enhancement of LLVM's BasicAA to the new interface. It reasons about `null` pointers, pointer casts, Φ-nodes, `select` instructions, and address computations, and asserts that stack allocations do not alias global variables.

Auto-restrict analyzes aliasing between formal parameters by considering all call sites (when known), and replacing formal parameters with actual parameters. In effect, this adds C99's `restrict` keyword to formals when appropriate.

Global Malloc identifies the subset of global variables for which all definitions (stores) may be exhaustively enumerated. It identifies subsets of globals for which (a) all definitions are from an allocator and (b) no definitions are from an allocator. It asserts that values loaded from globals in classes (a) and (b) must be disjoint.

No-Capture Global identifies the subset of global variables whose address is never captured. Using a reachability argument, it asserts that a pointer loaded from memory cannot alias with a non-captured global.

No-Capture Source identifies global variables or allocators whose address is never captured. Such objects can only be referenced through addresses computed from the object's name. The algorithm enumerates, transitively, all uses of that object, and thus may assert the disjointedness of these objects from other pointers.

Φ-Maze traces the definitions of pointers through Φ-nodes, pointer casts, and address computations to identify a set of allocation instructions. If those sets of source objects are complete, it then compares the sets, reasoning that disparate allocations are disjoint from one another, and from formal parameters or global variables.

Disjoint Fields identifies aggregate types which are never type-punned. It identifies within them fields whose address is never captured. All stores to such fields can be enumerated and collected as points-to sets. The algorithm asserts that pointers loaded from such fields are among the points-to sets, and reports that loaded pointers are disjoint if their points-to sets are disjoint.

4.2.2 Functor Analysis Algorithms

Functor algorithms initiate collaboration by generating premise queries and asking other algorithms to solve them.

AoS is the example from Section 3.1. Given indexing operations `&a[`i_1`]...[`i_n`]` and `&b[`j_1`]...[`j_n`]`, it reports no dependence if arithmetic and induction variable reasoning proves $i_k \neq j_k$ and if topping establishes $a = b$.

Kill-Flow searches for killing operations along all feasible paths between two operations. It searches blocks which post-dominate the source and dominate the destination. If it finds call sites among those blocks, it recurs into the callees. It tops premise queries to compare the footprints of potential killing operations, reporting no dependence if proved. Although worst-case running time is high, it runs quickly in practice due to careful evaluation order and caching.

Call Site Depth Combinator deconstructs queries between call sites into queries between operations within the callees. It issues premise queries for all memory operations within the call site. When recurring into a call site, it records calling context and tests kills (per Kill-Flow) along each level of context.

Semi-Local Function uses knowledge of standard C, POSIX, and C++ STL routines to identify user-defined functions which only read or write memory reachable through parameters or explicit lists of global variables. The algorithm generates premise queries to compare call site arguments.

Unique Access Paths searches for global, heap, or stack objects whose address is never captured. All stores to such objects can be enumerated, i.e. the memory objects have a unique access path. It collects points-to sets of values which are stored to those objects. The algorithm converts queries on pointers loaded from such paths into premise queries among the values in the points-to sets.

Field Malloc identifies aggregate types which are never type-punned and fields of such types whose address is never captured. It determines whether all stores to the fields are the unique use of an allocation, i.e., loading the field is the only way to reach the allocation. It asserts that pointers loaded from such fields alias only if a premise query demonstrates that their base objects alias.

5. Evaluation

We evaluate 13 analysis algorithms in the proposed framework in the LLVM Compiler Infrastructure [17] (revision 164307). We evaluate against 19 C and C++ applications from the SPEC 2006 suite [31]. We exclude eight benchmarks for lack of a FORTRAN front-end. Before evaluation, we optimize each benchmark with `clang -O3`, internalization, and devirtualization of indirect calls. We evaluate performance on *hot loops*, those which comprise at least 5% of execution time and iterate at least 5 times.

5.1 Methodology: Clients and Metrics

We follow Hind's recommendation [12] by evaluating with respect to clients of interest: a *PS-DSWP* client and a Program Dependence Graph (PDG) client.

The PS-DSWP client queries analysis to drive Parallel-Stage Decoupled Software Pipelining [29]. PS-DSWP schedules the Strongly Connected Components (SCCs) of the PDG across threads to produce a pipeline execution model. We use a fast algorithm to compute SCCs [14]. Several metrics of PS-DSWP are:

- `%NoDep`: percent of dependence queries for which the ensemble reports no flow, anti, or output dependence. Larger %NoDep is consistent with higher precision.
- `NumSCCs`: number of SCCs in the loop's PDG. More SCCs grant PS-DSWP greater scheduling freedom. Imprecise dependence analysis conservatively merges SCCs.
- `NumDoallSCCs`: number of SCCs which lack loop-carried dependences. More is better as PS-DSWP schedules DOALL SCCs concurrently.
- `%BailOut`: percent of loops for which the SCC algorithm bails out [14]. Bail-out indicates that PS-DSWP cannot parallelize the code. Smaller is better.

The PDG client [8] performs an intra-iteration and a loop-carried dependence query on each pair of memory operations within each hot loop. The PDG client's `%NoDep` metric is the fraction of queries which the ensemble disproves or reports only a read-after-read dependence. Larger `%NoDep` is better. This metric values every dependence equally.

Both clients are limited to a 30 minute timeout. In the case of a timeout, results indicate progress before a timeout.

5.2 Methodology: Dependence Profiling as an Oracle

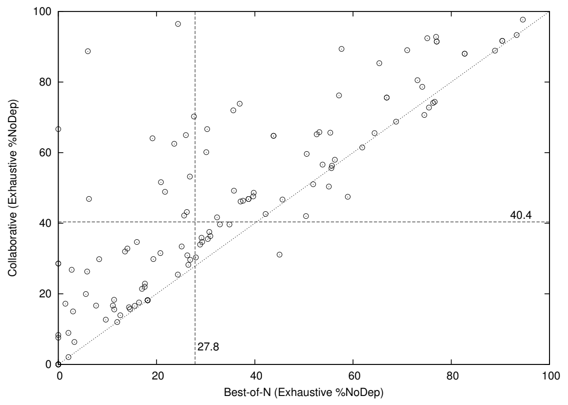
We model an oracle from profiling information. We use a loop-sensitive memory flow dependence profiler [25] to identify dependences in applications with its *spec_train* input set. If the profiler observes no memory flow dependence between a pair of operations, the oracle asserts that there is no flow dependence. An analysis-adapter introduces these assertions into the ensemble.

This is not a *true* oracle. Profiles are incomplete because the training input does not induce 100% code coverage. The memory profiler detects only *flow* dependences and cannot assert the absence of anti or output dependences. In these cases, the oracle degrades to static analysis.

In some cases, the oracle is *too* precise because profiling information reflects input-driven program behaviors, whereas static analysis algorithms compute a conservative estimate of program behavior over *all* possible inputs. We do not expect static analysis to achieve oracle results. Despite limitations, this oracle provides a reference for comparison.

5.3 Importance of Context

To evaluate the impact of context and path qualifiers on precision, we create variants of the PDG and PS-DSWP



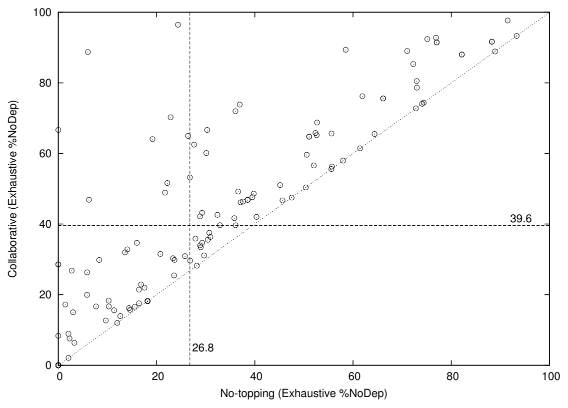


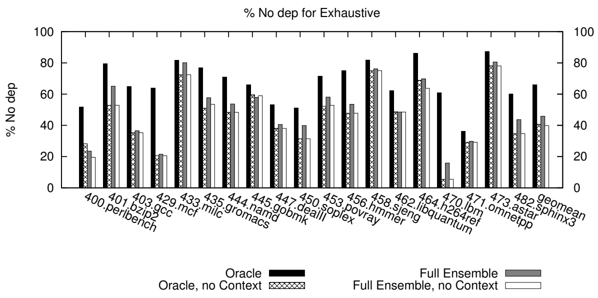
Figure 4: Relative strengths of composition methods. Each point is a hot loop, excluding time-outs. Cross-hairs show the geometric mean of each dimension, excluding loops with either x=0% or y=0%. Collaboration performs better on loops above the diagonal. *(above)* Collaboration vs best-of-*N*. *(below)* Collaboration vs no-topping.

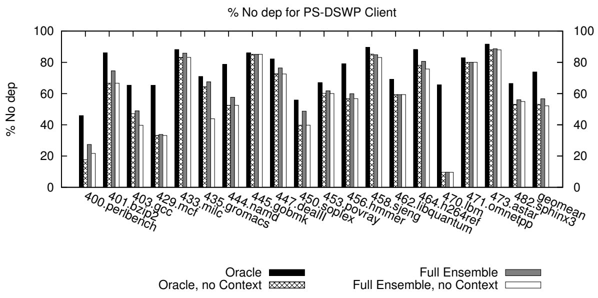
clients which issue context-blinded queries. We compare the performance of the contextualized and context-blinded variants of the oracle and the full ensemble of static analysis.

Figure 5 compares the precision of the oracle and full ensemble with and without query context. The top plot presents the %NoDep metric of the PDG client. When context is removed, oracle performance drops by 25.3% (geomean) and ensemble performance drops by 6.1% (geomean). The middle presents results with respect to the %NoDep metric of the PS-DSWP client which show a similar decrease in precision. The bottom plot shows the fraction of loops within each benchmark for which PS-DSWP bails out (finds no DOALL SCCs). Removing context reduces precision and increases the bail-out rate by 29.7% for the Oracle and 40.0% for the ensemble, indicating degraded client performance.

5.4 Precision with Respect to the PS-DSWP Client

Of the 169 hot loops found in 19 SPEC 2006 benchmarks, 20 loops are so constrained by register and control dependences that they have only one SCC. The PS-DSWP client bails-out before it issues any memory query for such loops. Of the 149 hot loops for which PS-DSWP issues queries, the oracle reports worst-case (1 SCC, 0 DOALL SCCs) for 84 hot loops (56.4%). The full ensemble reports the same in these cases.





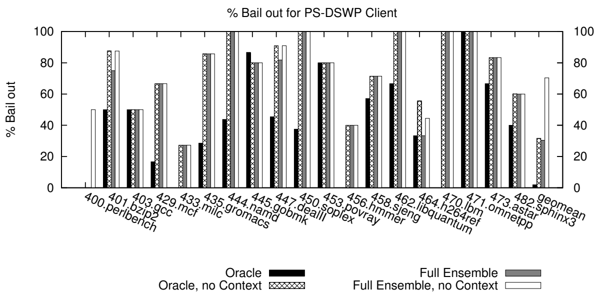


Figure 5: Query context improves oracle and full ensemble precision. *(top)* The PDG Client's %NoDep metric. *(middle)* The PS-DSWP Client's %NoDep metric. *(bottom)* The PS-DSWP Client's bail-out metric.

Among 65 remaining loops, the oracle found more than one SCC. On these loops, the full ensemble reported 27.4% as many SCCs as the oracle (geomean). The full ensemble achieves zero DOALL SCCs for 39 loops. Excluding these loops, the full ensemble reports 66.5% of the DOALL SCCs of the oracle (geomean).

Overall, the PS-DSWP client reports the same number of SCCs for 97 of 169 hot loops, and reports the same number of DOALL SCCs for 98 hot loops, regardless of whether the client employs the oracle or full ensemble.

5.5 Topping, Chaining, or Best-of-N

The proposal encourages development of factored algorithms, arguing that the pattern of chaining and topping achieves precision. We demonstrate the value of chaining and topping by evaluating alternative means of composition while using the same algorithms.

An alternative to chaining and topping is the *best-of-N* method which passes each query to each algorithm in isolation and returns the most precise answer. This corresponds to the implementation in the LLVM static analysis framework

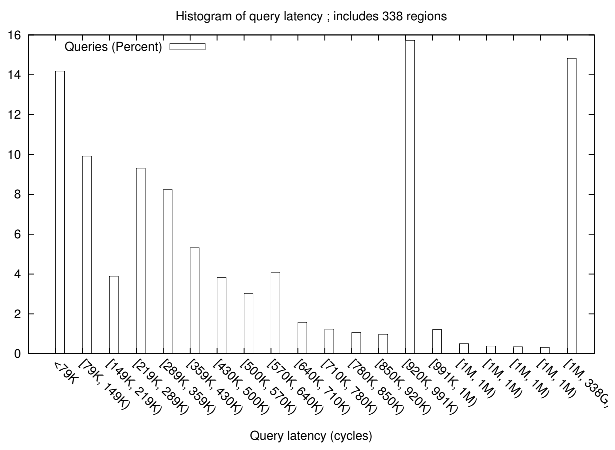


Figure 6: Histogram of query latency.

Gain = 1/3 = Isolated $\implies$ Orthogonal

	Q1	Q2	Q3	%No
All algorithms	No	No	–	2/3
Algorithm *X*, isolated	–	No	–	1/3
All algorithms except *X*	No	–	–	1/3

Gain = 0/3 < Isolated $\implies$ Non-orthogonal

	Q1	Q2	Q3	%No
All algorithms	No	–	–	1/3
Algorithm *X*, isolated	No	–	–	1/3
All algorithms except *X*	No	–	–	1/3

Gain = 2/3 > Isolated $\implies$ Collaboration

	Q1	Q2	Q3	%No
All algorithms	No	No	No	3/3
Algorithm *X*, isolated	–	No	–	1/3
All algorithms except *X*	No	–	–	1/3

Figure 7: Different ensembles disprove different subsets of a fixed set of queries. By varying the ensemble, we observe orthogonality, non-orthogonality, and collaboration.

(see Section 6). Figure 4 compares the best-of-*N* method to our proposal. The chaining and topping method performs better than best-of-*N* on the %NoDep metric of the PDG client for all but 10 of 140 loops (timeouts excluded). Precision bugs in our implementation cause these cases.

To evaluate the importance of premise queries, we modified our framework so that every topped query is instead chained, i.e., premise queries are passed only to later members of the ensemble instead of all. Figure 4 presents this comparison and demonstrates that chaining and topping combined dominate chaining alone.

5.6 Query Latency

Figure 6 presents the time to service queries from the SPEC 2006 suite. Time is measured in processor cycles on an eight core 1.6GHz Xeon E5310. Measurements are performed with the full ensemble, as well as a few additional analyses not described in this paper due to space considerations. Overall, 50% of queries are serviced in 287.6μs (460K cycles) and 90% of queries are serviced in 1.0ms (2M cycles). Variations in query latency are due to variations in query complexity, not noise.

5.7 Collaboration

Evaluation demonstrates that factored algorithms work together despite strict modularity. For a fixed set of dependence queries, such as the queries necessary to compute the PDG of a loop, an algorithm's *orthogonality* and *collaboration* can be witnessed as the marginal improvement when adding or removing that algorithm from the ensemble.

Figure 7 illustrates this methodology with a fixed set of three queries. The *full ensemble* contains all 13 algorithms. We define *leave-one-out* and *isolated* ensembles for each algorithm *X*: leave-one-out consists of all algorithms except for *X*, and isolated consists of *X* alone. We define gain as the difference between the full ensemble and the leave-one-out ensemble of *X*. Informally, gain represents the contribution of algorithm *X* to the ensemble. Algorithms are *orthogonal* if at most one algorithm disproves any one dependence query: adding *X* increases the ensemble's precision by the precision of *X* in isolation. Algorithms are *non-orthogonal* if there is a query which several disprove: adding *X* improves precision by an amount less than the precision of *X* in isolation. We say that algorithms in an ensemble *collaborate* if there is a class of dependence queries which the ensemble disproves, yet which no single algorithm disproves in isolation.

We value orthogonality as it is consistent with the minimal amount of software development effort, but non-orthogonality is not detrimental to soundness or precision. We seek collaboration since it indicates a great return on software-engineering effort.

Table 3 summarizes collaboration and orthogonality experiments by comparing full ensemble, leave-one-out, and isolated performance of each algorithm. Columns present the percentages of loops whose gain is greater than, equal to, or less than its isolated performance. Each loop is an aggregation of queries, so all categories potentially represent a mixture of collaboration, anti-collaboration, orthogonality and non-orthogonality. When gain exceeds isolated, the algorithm contributes more in an ensemble than it does on its own. Such loops are positive evidence of collaboration. Loops whose gain equals isolated performance are consistent with orthogonality. Loops whose gain is zero indicate non-orthogonality. Loops whose gain is less than isolated performance are inconclusive.

We present loop-by-loop data for four select algorithms. The comparisons in Table 3 correspond to the position of each loop with respect to the diagonals in Figure 8. Combinator algorithms Array of Structures and Kill Flow show a trend along the isolated=0 border, indicating that these algorithms disprove few queries alone yet help other algorithms to disprove many. Basic Loop demonstrates many loops with isolated>0. However, it also demonstrates collaborative loops where gain>isolated. Although it does not generate premise queries, Basic Loop collaborates by solving premise queries generated by other analyses.

There may be a class of dependences which one algorithm disproves in isolation, but which the ensemble cannot. We call such cases *anti-collaborative*. Developers try to avoid anti-collaboration since it indicates a precision bug. Table 3 and Figure 8 show some analysis implementations suffer from anti-collaboration, visible as negative gain. Ar-

Analysis Algorithm	Gain vs Isolated Performance (% of Loops)				
	Collab. gain > iso	*Ortho?* gain = iso	*Anti-collab.* gain < 0 < iso	*Non-ortho.* 0 = gain < iso	*Anti-collab? Non-ortho?* 0 < gain < iso
Kill flow	40.7	44.3	2.1	3.6	9.3
Callsite depth combinator	32.1	57.1	0.0	0.0	10.7
Array of structures	31.4	50.0	13.6	0.7	4.3
Semi local fun	30.9	50.4	1.4	2.9	14.4
Basic loop	30.2	24.5	0.7	8.6	36.0
Field malloc	21.4	72.1	2.9	2.1	1.4
Unique access paths	13.7	86.3	0.0	0.0	0.0
Auto restrict	10.9	84.1	5.1	0.0	0.0
No capture global	6.4	57.9	3.6	15.7	16.4
Global malloc	1.4	89.9	4.3	2.9	1.4
Phi maze	1.4	56.8	2.9	28.8	10.1
No capture src	0.0	63.3	2.9	20.9	12.9
Disjoint fields	0.0	82.9	5.0	5.0	7.1

Table 3: Collaboration and anti-collaboration is observed in the relative strengths of the full, leave-one-out, and isolated ensembles. Each cell is a percentage of hot loops which satisfy an inequality between gain and isolated ensembles.

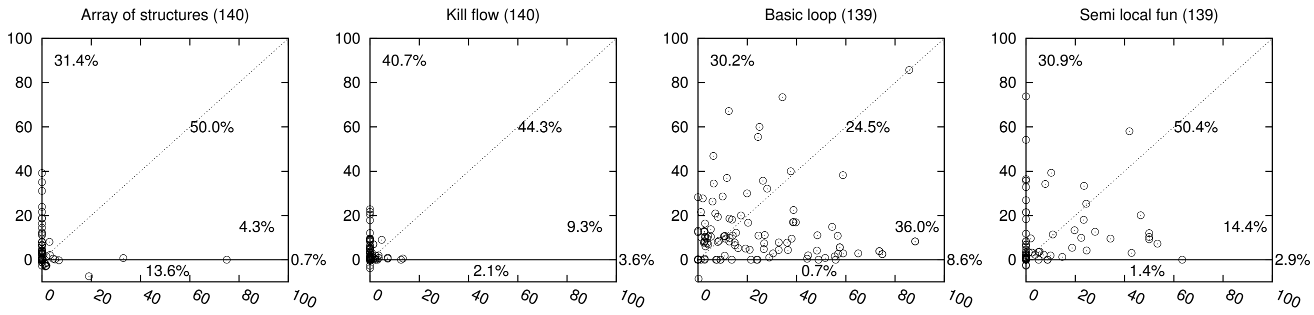


Figure 8: Collaboration per loop of the PDG %NoDep metric for select algorithms. The horizontal axis is isolated performance, the vertical axis is gain (full - leave-one-out). Percentages in plot area count loops in each case.

ray of structures, for instance, shows anti-collaboration on 13.6% of loops, but most of those are only slightly negative.

5.8 Importance to Speculative Parallelization

The ASAP System [15] is a speculative automatic thread extraction system for clusters. ASAP combines static analysis and profile-driven speculation to identify and parallelized hot loops. Its runtime system provides a transactional memory abstraction and provides memory coherence between the otherwise disjoint memories at each cluster node.

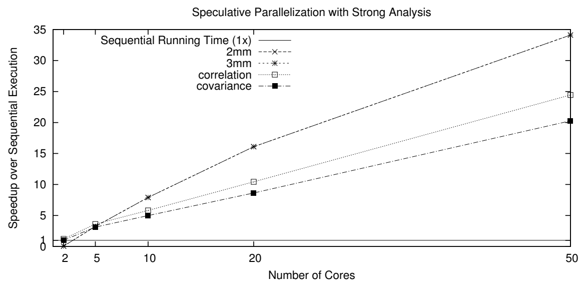
ASAP uses speculation to overcome weak static analysis, but speculation incurs high runtime costs in terms of validation overheads. When more precise analysis is available, the compiler generates more efficient code with lower overheads, and eliminates validation overheads completely when non-speculative parallelization is possible.

To analyze the sensitivity of a speculative thread extraction system to dependence analysis precision, we repeat the ASAP evaluation while varying the strength of dependence analysis. Each analysis configuration employ a subset of all analysis implementations. We select 28 analysis configurations: (a) the *full* configuration of all analyses; (b) 13 *leave-one-out* configurations, each consisting of the full set of analyses with one removed; (c) 13 *singleton* configurations, each consisting of a single analysis in isolation; (d) the *null* configuration with no analyses. We expect configuration (a) to be most precise, followed by configurations in (b), configurations in (c), and finally configuration (d). We ran the ASAP automatic thread extraction system under each configuration of analysis. Although the compiler could produce 28 distinct variants of each benchmark, each corresponding to an analysis configuration, in practice the compiler generated at most two distinct outputs per benchmark.

Seven benchmarks were insensitive analysis precision, two crashed, and four were sensitive. Figure 9 summarizes the results of the ASAP analysis-sensitivity experiments for the benchmarks `2mm`, `3mm`, `covariance` and `correlation`. With stronger analysis, ASAP achieves a geomean speedup of 28× on 50 cores, however, when weak analysis is used, those same benchmarks suffer an 11× slowdown. All slowdown measurements are bound by a 24 hour timeout.

6. Related Work

The LLVM compiler infrastructure [17] uses ensembles which combine analysis algorithms with chaining. LLVM implements classic algorithms [1, 32] and state of the art algorithms (such as [18]). However, the LLVM interface lacks context and a means to “top” premise queries. The LLVM query language’s lack of context limits its precision and leads to separate implementations `MemoryDependenceAnalysis` and `LoopDependenceAnalysis`. Each is a monolithic implementation, not a collaborative or even composable interface, limiting future development.



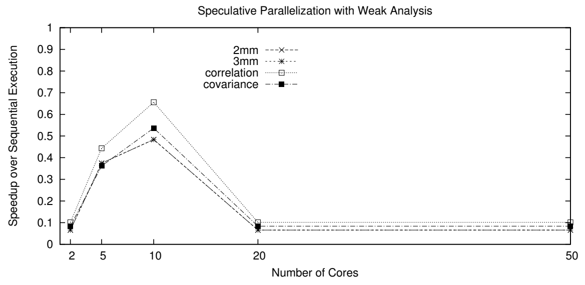


Figure 9: Sensitivity of ASAP speedup to the static analysis. In four benchmarks, stronger analysis allows parallelization without speculation or with less speculation, and delivers scalable parallelism. Weaker analysis forces the compiler to employ more speculation on the same benchmarks, and parallelized applications run *slower* than the sequential code.

LLVM design documents [24] hint at premise queries. However none of the LLVM implementations use premises (Basic, Lib Calls, Globals ModRef, Scalar Evolution, Type-Based, Steensgaard's, nor Data-Structure Analysis). Instead, these algorithms chain inapplicable queries verbatim. Without premises, LLVM follows the *best-of-N* composition pattern (Section 5.5).

Even if these algorithms were designed to generate premise queries, LLVM's analysis framework lacks a topping mechanism. Design documents suggest placing algorithms which issue premise queries at higher positions in the ensemble so that chained premise queries reach more, subsequent implementations [24]. This suggestion resembles *no-topping* composition (Section 5.5). Since our proposal lets functors top queries regardless of their positions, developers may adjust scheduling priorities (Section 3.2.1) to tune query latency without harming precision.

The 2013 Dagstuhl Seminar considers pointer analysis [22]. The report recognizes the importance of qualifying analysis queries with context. Our query language's loop context and temporal relation (Section 3.3.2) provides a concrete interface to achieve this. The report also suggests dividing the interface: one interface between client and analysis systems, and another between the analysis system and member algorithms. The former translates IR-specific queries into a canonical form and allows member algorithms to post intermediate results. A central query system combines these intermediate results as a means of collaboration. Our proposal instead uses one interface for both roles. Our proposal includes a query language that allows context to be expressed by both clients and other analysis algorithms. Due to the similarities among dependence, alias, and shape analysis, our query language is expressive enough for algorithms to exchange rich premise and collaborate despite incompatibilities among models that algorithms use internally.

Others examine collaboration between a limited number of specific analyses [3, 4, 20, 22]. For example, Thresher [3] combines pointer analysis with shape analysis using a mixed representation to enable collaboration. However, this collaboration is custom-built around these algorithms; it is unclear how it generalizes to other analysis algorithms.

Several Datalog frameworks implement analysis algorithms with various abstractions [5, 21, 33]. These allow clients to select a precision-scalability compromise by fixing an abstraction, but do not combine different abstractions.

Other works achieve precision and scalability by combining fast, imprecise analysis algorithms with slow, precise algorithms. Client-driven approaches track imprecision from polluting assignments in a fast algorithm and reissue queries to a precise algorithm when imprecision detriments the client [11]. The pruning-refinement method constructs a family of algorithms with varied abstractions [23]. A query visits each algorithm until one answers definitively. Intermediate results prune extraneous relations in later algorithms and improve scalability. Pruning does not improve any member algorithm's precision; thus they do not collaborate.

Composing analyses to achieve better dataflow optimization results has been explored in the past [6, 19]. Similar to our framework, these works are driven by the desire for simple and modular, rather than monolithic, analyses. Other works [7, 16, 27] contain a notion of collaboration between many analyses, however they focus on abstract or specialized domains such as SMT solvers. None of these techniques readily lend themselves to memory analysis.

7. Conclusion

The proposed framework enables collaboration among analysis algorithms in a composable framework. These features encourage the development of factored algorithms, whereby compiler engineers develop simple algorithms targeting orthogonal concerns. The framework's interface supports contextualized queries to disambiguate subtly different queries. Using an ensemble of many simple analysis implementations, the framework achieves precise dependence analysis to support advanced compiler optimizations. Furthermore, we believe that the core idea of a collaborative analysis framework could be extended beyond program dependence analysis to other types of program analysis.

8. Acknowledgments

We thank the Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Science Foundation (NSF) through Grants OCI-1047879, CCF-1439085, and CNS-0964328. All opinions, findings, conclusions, and recommendations expressed in this paper are those of the Liberty Research Group and do not necessarily reflect the views of the NSF. This work was carried out when the authors were working at Princeton University.

References

[1] L. O. Andersen. Program analysis and specialization for the C programming language, May 1994.

[2] U. Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, Boston, MA, 1994.

[3] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Thresher: precise refutations for heap reachability. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 275–286, New York, NY, USA, 2013. ACM.

[4] M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 1–12, New York, NY, USA, 2009. ACM.

[5] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[6] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17, 1995.

[7] P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *Proceedings of the 14th International Conference on Foundations of Software Science and Computational Structures: Part of the Joint European Conferences on Theory and Practice of Software*, FOSSACS'11/ETAPS'11, pages 456–472, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[9] R. Ghiya and L. J. Hendren. Is it a Tree, DAG, or Cyclic Graph? In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1996.

[10] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–265, June 2007.

[11] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *In International Static Analysis Symposium*, pages 214–236. Springer-Verlag, 2003.

[12] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[13] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1), January 1997.

[14] N. P. Johnson, T. Oh, A. Zaks, and D. I. August. Fast condensation of the program dependence graph. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 39–50, New York, NY, USA, 2013. ACM.

[15] H. Kim. *ASAP: Automatic Speculative Acyclic Parallelization for Clusters*. PhD thesis, Princeton, NJ, USA, 2013.

[16] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 1–12, New York, NY, USA, 2005. ACM.

[17] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[18] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, June 2007.

[19] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 270–282, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9.

[20] O. Lhotak. *Program Analysis using Binary Decision Diagrams*. PhD thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada, January 2006.

[21] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, Oct. 2008.

[22] O. Lhoták, Y. Smaragdakis, and M. Sridharan. Pointer Analysis (Dagstuhl Seminar 13162). *Dagstuhl Reports*, 3(4): 91–113, 2013. URL `http://drops.dagstuhl.de/opus/volltexte/2013/4169`.

[23] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[24] LLVM Project. LLVM Alias Analysis Infrastructure, October 2013. http://llvm.org/docs/AliasAnalysis.html.

[25] T. R. Mason. Lampview: A loop-aware toolset for facilitating parallelization. Master's thesis, Department of Electrical Engineering, Princeton University, Princeton, New Jersey, United States, August 2009.

[26] R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analyses. In *In Proc. ACM Symp. on Principles of Programming Languages*, pages 67–80. ACM Press, 2000.

[27] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, 1979.

[28] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing 1991*, pages 4–13, November 1991.

[29] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2008.

[30] M. Sagiv, T. Reps, and R.Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, January 1996.

[31] spec. Standard Performance Evaluation Corporation. http://www.spec.org.

[32] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[33] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, (PLDI), pages 131–144, New York, NY, 2004.