
# SCAF: A Speculation-Aware Collaborative Dependence Analysis Framework

Sotiris Apostolakis
Princeton University, USA

Ziyang Xu
Princeton University, USA

Zujun Tan
Princeton University, USA

Greg Chan
Princeton University, USA

Simone Campanoni
Northwestern University, USA

David I. August
Princeton University, USA

## Abstract

Program analysis determines the potential dataflow and control flow relationships among instructions so that compiler optimizations can respect these relationships to transform code correctly. Since many of these relationships rarely or never occur, speculative optimizations assert they do not exist while optimizing the code. To preserve correctness, speculative optimizations add validation checks to activate recovery code when these assertions prove untrue. This approach results in many missed opportunities because program analysis and thus other optimizations remain unaware of the full impact of these dynamically-enforced speculative assertions. To address this problem, this paper presents SCAF, a Speculation-aware Collaborative dependence Analysis Framework. SCAF learns of available speculative assertions via profiling, computes their full impact on memory dependence analysis, and makes this resulting information available for all code optimizations. SCAF is modular (adding new analysis modules is easy) and collaborative (modules cooperate to produce a result more precise than the confluence of all individual results). Relative to the best prior speculation-aware dependence analysis technique, by computing the full impact of speculation on memory dependence analysis, SCAF dramatically reduces the need for expensive-to-validate memory speculation in the hot loops of all 16 evaluated C/C++ SPEC benchmarks.

***CCS Concepts*** • **Software and its engineering → Compilers**; **Automated static analysis**; **Dynamic analysis**.

***Keywords*** speculation, dependence analysis, collaboration


*PLDI '20, June 15–20, 2020, London, UK*

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386028

**ACM Reference Format:**
Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I. August. 2020. SCAF: A Speculation-Aware Collaborative Dependence Analysis Framework. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3385412.3386028

## 1 Introduction

Program analysis allows compiler optimizations to transform code while respecting data and control flow relationships between instructions. Increased program analysis precision can dramatically improve the effectiveness of compiler optimizations, including those that perform instruction-level parallelization (ILP), thread-level parallelization (TLP), and vectorization. Thus, decades of research have been devoted to increasing the precision of program analysis. Advancements include algorithms in points-to analysis [1, 4, 6, 33, 35, 52], alias analysis [37, 59], shape analysis [19, 20, 50], and loop dependence analysis [3, 45]. Nevertheless, program analysis is undecidable [31] and remains insufficiently precise in practice, especially for languages like C/C++ [21].

Speculation allows optimizations to overcome the limitations of program analysis. Speculation typically relies on profile-based information to identify data and control flow relationships expected to rarely or never occur during program execution. Speculative optimizations optimize for the common case by assuming that these relationships do not exist while transforming the code. To preserve correctness, speculative optimizations add checks to activate recovery code when these assumptions prove untrue. To be profitable, speculative optimizations must consider the benefits of optimizing for the common case against the expected frequency of misspeculation, the cost of misspeculation recovery, and the validation cost. The validation cost is the cost of checking for misspeculation, a cost that exists even when there is no misspeculation. Note that relationships reported by program analysis but not observed during profiling may actually exist (analysis is not limiting) or not exist (analysis is imprecise). In either case, the benefits of speculation remain.

The validation and recovery code inserted by speculative optimizations can be viewed as dynamically-enforced assertions ensuring that certain relationships reported by program analysis cannot exist in the protected code. In existing compiler designs, subsequent program analysis and optimization passes operate on the transformed code, unaware of the full impact of speculative assertions. This is problematic because the unrecognized value of a single speculative assertion can be significant. For example, the application of control speculation to speculatively enforce the elimination of a control path may make many previously reported memory dependences impossible. Unaware of the speculative control flow information, the compiler will needlessly continue to respect the now nonexistent memory dependences. This might lead to the compiler unnecessarily preventing the application of valuable transformations on account of the phantom memory dependences. Alternatively, it might lead to the pointless application of additional speculation to remove the phantom memory dependences. Even worse, this additional speculation is typically much more expensive than the already-applied control speculation.

Speculative optimizations themselves represent a breakthrough that has helped overcome the crippling limitations of program analysis, especially for ILP and TLP compilers [25, 61]. Nevertheless, many authors report that speculative systems often suffer from large overheads [7, 16], most notably the overheads from the cost of checks and the overly-aggressively application of speculation. This cost is particularly acute for memory dependence speculation because of the large number of memory dependences reported by analysis that do not manifest during profiling and because of the high validation cost of memory speculation for each speculatively removed dependence [7, 16, 55]. As this work demonstrates, much of this cost is the result of the lack of speculative assertion awareness in compiler analysis and optimization.

The goal of this work is to enable lower-cost speculation with a modular, collaborative, and speculation-aware memory analysis framework. This speculation-aware collaborative dependence analysis framework, called SCAF, learns of available speculative assertions, computes their full impact on memory dependence analysis, and makes this resulting information available for code optimization. In this way, SCAF enables the compiler to make the most of speculation by speculating more judiciously. Like the collaborative analysis framework (CAF [24]) of prior work, SCAF is modular and collaborative. The modularity makes the addition of new analysis modules easy. The collaborative aspects mean that analysis modules cooperate to produce a result more precise than the confluence of all individual results. Relative to CAF, SCAF is made possible *(i)* by the addition of speculation modules, a new type of analysis module that uses profiling information to answer queries; *(ii)* by the introduction of a new coordinating component called the *Orchestrator*; and, *(iii)* by extensions to CAF's dependence analysis query language and its semantics to carry additional information related to speculation.

This paper:

- introduces SCAF, the first modular, collaborative, and speculation-aware dependence analysis framework;
- motivates and describes SCAF's design (§2,§3);
- demonstrates how SCAF enables existing speculation-unaware memory analysis modules to reason about speculation (§3);
- describes query language extensions to support communication of control flow, in addition to data flow, information and to reduce query latency (§3.2.2);
- introduces a new compiler component, called the *Orchestrator*, that coordinates interactions among analysis modules and is configurable according to the client's preferences (§3.3);
- presents a design pattern for speculation modules in a collaborative environment (§4.2.1);
- describes speculation modules implemented in SCAF (§4.2.3, §4.2.4); and,
- evaluates SCAF on 16 C/C++ SPEC benchmarks, and demonstrates its ability to decrease the need for expensive-to-validate memory speculation by maximizing the impact of inexpensive speculation (§5).

## 2 Background & Motivation

This section defines what a memory dependence is, discusses the role of speculation in memory dependence analysis, and motivates the need for a collaborative, modular, and speculation-aware dependence analysis framework.

### 2.1 Memory Dependence

A memory dependence from instruction $i_1$ to instruction $i_2$ exists iff: (i) the *footprint* of operation $i_1$ *may-alias* the *footprint* of $i_2$ (*alias*); (ii) at least one of the two instructions writes to memory (*update*); (iii) there is a feasible path of execution $P$ from $i_1$ to $i_2$ (*feasible-path*) such that (iv) no operation in $P$ overwrites the common memory *footprint* (*no-kill*). *Footprint* refers to the memory locations accessed (read or written) by an instruction.

### 2.2 Speculation

A diverse set of speculation techniques have been proposed to overcome the imprecision of memory analysis [8–10, 18, 25, 28, 44, 53, 54, 56, 58, 60]. Aggressive use of speculation is most prominently observed in parallelization schemes [25, 28, 40, 47, 56], where high performance gains can compensate for overheads introduced by speculation.

To estimate data and control flow relationships among instructions, speculative techniques typically rely on profiling information. Offline runs of the target program with representative inputs produce this profiling information. We refer

**Table 1.** Comparison of Proposals for Integration of Speculation into Analysis

| Approaches | Supported Forms of Collaboration | | Memory Analysis Decoupled from Speculation |
|---|---|---|---|
| | Among Speculative Techniques | Between Memory Analysis and Speculative Techniques | |
| Monolithic Integration [2, 12, 14] | ✗ | ✓ | ✗ |
| Composition by Confluence [28, 40, 57, 61] | ✗ | ✗ | ✓ |
| Composition by Collaboration (**This Work**) | ✓ | ✓ | ✓ |

to predictions based on profiling information as *speculative assertions*.

### 2.2.1 Express Impact of Speculation

This section describes how prior work expresses the impact of *speculative assertions* to other transformations and analyses within the compiler, and then motivates the approach proposed in this paper.

One might attempt to express the effect of speculative assertions by transforming the code. For example, Neelakantam et al. [41] propose converting biased branches to assertions to expose speculative control flow information to subsequent transformations. This approach, though, does not generalize for all the types of speculative assertions. For example, the impact of separation speculation [25] and memory speculation cannot be expressed via a transformation. Further, applying speculative transformations without fully evaluating their enabling effect is problematic. Compilers should only apply speculative transformations that enable optimizations with performance gains exceeding the speculation overheads. Moreover, the application of a transform may limit the applicability of some subsequent transforms (phase order problem).

To avoid these pitfalls, the impact of speculative information needs to be visible during an analysis phase prior to transformation. This requires integrating speculation into memory analysis. Prior work has explored two different ways to perform this integration: via *composition by confluence* and *monolithically*.

***Composition by Confluence:*** To integrate speculative information into an analysis phase, some consider the effect of speculative techniques on memory dependences in a sequence, independently of each other and of memory analysis [28, 40, 57, 61]. We characterize this approach as *composition by confluence* since the result of this composition is the confluence of all individual techniques' results. This design is modular (consists of independently developed components), but does not support the synergistic co-existence of speculation and analysis. Therefore, it fails to fully leverage the impact of speculative information, as shown in the motivating example (§2.2.2).

***Monolithic Integration:*** In this approach, memory analysis algorithms are extended with knowledge and interpretation of profile-based speculative information [2, 12, 14]. This scheme increases the impact of speculative assertions. Yet, given the diverse set of existing memory analysis algorithms and speculative techniques, creating monolithic and complex implementations of different combinations does not scale and hinders extensibility and maintainability.

***Composition by Collaboration:*** Motivated by the deficiencies of prior work, this paper introduces a new approach of integrating speculation with memory analysis. The proposed approach exposes the full impact of speculative assertions by enabling collaboration of memory analysis and speculative techniques (*composition by collaboration*) without sacrificing modularity and prior to any transformation. In fact, this scheme allows memory analysis to leverage speculative information despite being developed independently.

Table 1 summarizes the comparison of this work with existing proposals of the designs described earlier in this section.

### 2.2.2 Motivating Example

For the code example in Figure 1, a client wants to determine whether there is a cross-iteration dependence from instruction $i3$ to $i2$. By inspecting the code, one can observe that there is a cross-iteration data flow from $i3$ to $i2$ when the branch is taken. However, since this path is highly unlikely to execute, one could speculatively ignore it and infer that instruction $i1$ kills the data flow from $i3$ to $i2$.

A compiler using memory analysis cannot disprove the cross-iteration data flow from $i3$ to $i2$ since none of the conditions described in §2.1 (*alias*, *update*, *feasible-path*, *no-kill*)

```
1 loop L:
2     if (rare)
3         // no writes to a
4         ...
5     else
6 i1:     a = ...;
7 i2: b = foo(a);
8     ...
9 i3: a = ...;
```

**Figure 1.** Motivating Code Example

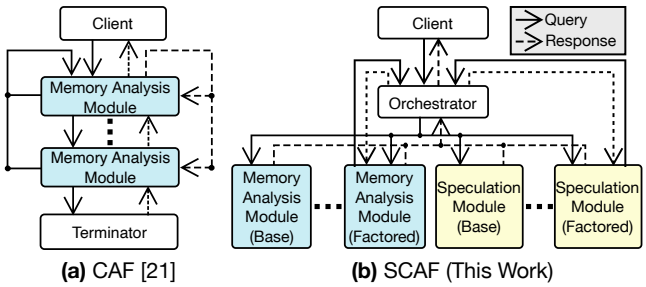



**Figure 2.** Design of Collaborative Analysis Frameworks

can be statically disproven. Further, control speculation cannot assert the absence of this dependence in isolation since neither $i2$ nor $i3$ are speculatively dead. Therefore, *composition by confluence* is unable to remove this dependence.

To maximize the impact of the control flow assertion (branch is never taken), interaction among control speculation and memory analysis is necessary in this example.

The *monolithic integration* approach would extend the kill-flow analysis algorithm [24] to interpret edge profiling information. This way, kill-flow can leverage the biased branch in our example, view $i1$ as executing on every iteration, infer that the condition *no-kill* from §2.1 is violated and thus can assert the absence of the cross-iteration data flow from $i3$ to $i2$.

Instead, this work is able to assert the absence of the cross-iteration data flow from $i3$ to $i2$ without any transformation and in a modular fashion as shown in §3.5.

# 3 SCAF

This work presents SCAF, a modular and collaborative dependence analysis framework that enables collaboration between memory analysis and speculative techniques (Figure 2b).

SCAF can be seen as a speculation-aware extension of CAF [24]. CAF is limited to collaboration among memory analysis algorithms, which are depicted as memory analysis modules in Figure 2. SCAF introduces speculation into the analysis framework with the introduction of *speculation modules*. Speculation modules express the effect of speculative techniques by interpreting profiling information in terms of dependence analysis.

## 3.1 Collaboration

Collaboration among modules in SCAF occurs indirectly through a new coordinating component, called the *Orchestrator*. Modules may formulate *premise queries* from incoming queries to resolve propositions about which they cannot reason (inspired by CAF [24]). Modules that create premise queries are called *factored* modules, while the rest are called *base* modules. Premise queries are sent back to the *Orchestrator* to allow other modules to resolve them, and effectively contribute to the resolution of the original queries. That way, modules are agnostic to who produces an incoming query or to who assists them, and there is no need for direct communication among modules. In fact, memory analysis modules, despite being *speculation-unaware*, can collaborate with speculation modules. This decoupled design enables independent development of modules and easy extension of the framework.

## 3.2 Query Language

The query language enables interactions between clients and analysis modules, and among the analysis modules. It defines how dependence analysis queries are expressed and serves as the modules' interface. Figure 3 defines the syntax of the analysis queries (§3.2.1, §3.2.2) and the query responses (§3.2.3).

### 3.2.1 Query Types

As in LLVM's alias analysis infrastructure (LLVM 5.0 [37]) and CAF [24], SCAF supports two types of analysis queries: `alias` and `modref` queries. Alias queries determine whether two pointers may alias each other, while modref queries determine whether an instruction may read or write a memory location (defined by a pointer and a location size) or the memory footprint of another instruction.

### 3.2.2 New Query Parameters

This paper introduces new query parameters, compared to CAF and LLVM, essential for collaboration in the presence of speculative analysis modules, and for query latency reduction.

In a traditional memory analysis framework, there is only one valid control flow graph. However, the introduction of speculation modules, particularly modules that interpret branch-related profile information, enables new variants of the control flow. To allow modules to communicate control-flow knowledge, we introduce optional *control-flow* query parameters in the form of dominator and post-dominator trees. This way, control-flow sensitive modules of the ensemble can leverage this speculative information to resolve queries, unresolvable with the traditional static control flow information. Even so, modules are agnostic to whether the control flow information contained in the received query is speculative or not.

Modules that generate premise alias queries often benefit from only one specific alias result. However, CAF's (and LLVM's) interface does not differentiate a must-alias query from a query that is meant to check no-alias. Therefore, this paper introduces another (optional) parameter that allows modules to specify exactly the alias result they need from premise alias queries to resolve the original query. This new parameter significantly reduces the query latency (§5.3) since modules can bail-out early if they cannot return the required answer.

**Query Syntax**

| | | |
|---|---|---|
| Query | $q$ | $::= q_a \mid q_m$ |
| Alias Query | $q_a$ | $::= \textbf{alias}(m_1, tr, m_2, l, cc, dr)$ |
| ModRef Query | $q_m$ | $::= \textbf{modref}(i, tr, m, l, cc, dt, pdt)$ |
| | | $\mid \textbf{modref}(i_1, tr, i_2, l, cc, dt, pdt)$ |
| Memory Location | $m$ | $::= (p, s)$ |
| Temporal Relation | $tr$ | $::= \textbf{Before} \mid \textbf{Same} \mid \textbf{After}$ |
| Desired Result | $dr$ | $::= \textbf{NoAlias} \mid \textbf{MustAlias}$ |

**Response Syntax**

| | | |
|---|---|---|
| Query Response | $r$ | $::= (\mathcal{R}, \mathcal{S})$ |
| Result | $\mathcal{R}$ | $::= \mathcal{R}_a \mid \mathcal{R}_m$ |
| Alias Result | $\mathcal{R}_a$ | $::= \textbf{NoAlias} \mid \textbf{MustAlias} \mid \textbf{SubAlias} \mid \textbf{MayAlias}$ |
| Modref Result | $\mathcal{R}_m$ | $::= \textbf{NoModRef} \mid \textbf{Ref} \mid \textbf{Mod} \mid \textbf{ModRef}$ |
| Set of Options | $\mathcal{S}$ | $::= \emptyset \mid \{O\} \mid \mathcal{S} + \mathcal{S} \mid \mathcal{S} \times \mathcal{S}$ |
| Assertion Option | $O$ | $::= \emptyset \mid \{\mathcal{A}\} \mid O + O$ |
| Assertion | $\mathcal{A}$ | $::= (id, tp, ec, cp)$ |

**Other Notations**

| | | | | | |
|---|---|---|---|---|---|
| $i$ : | Instruction | $cc$ : | Calling Context | $tp$ : | Transformation Points |
| $p$ : | Pointer | $dt$ : | Dominator Tree | $ec$ : | Estimated Cost |
| $s$ : | Access Size | $pdt$ : | Post-Dominator Tree | $cp$ : | Conflict Points |
| $l$ : | Loop | $id$ : | Module ID | | |

**Figure 3.** Syntax for SCAF's query and query response. Colored text indicates syntax extensions over the query language of non-speculative analysis frameworks (CAF [24], LLVM [37]). `Before`, `After`, and `Same` denote the first operation executes/the values of the first pointer are computed in a strictly-earlier/a strictly-later/the same iteration than/as the second.

Another introduced (optional) parameter provides *calling-context* information. This context helps disambiguate between different dynamic instances of the same static instruction. This parameter is essential for more fine-grained identification of memory objects, since several memory objects may be created by the same static instruction. Speculation analysis modules that reason about memory objects benefit from this context.

Moreover, queries in SCAF, same as in CAF [24], contain additional context information via the *loop* and *temporal relation* parameters. The loop parameter scopes the query to represent dynamic instances of operations during the loop's execution. The temporal relation restricts the considered paths and allows distinguishability between intra-iteration (`Same`) and cross-iteration (`Before`, `After`) dependences.

### 3.2.3 (Speculative) Query Response

Memory analysis frameworks [24, 37] do not need to provide any additional information apart from the query result (e.g., `NoAlias`). In SCAF, by contrast, answers might be predicated on speculative assertions that need to be validated at runtime if the client wishes to preserve the semantics of the original code. Thus, query responses in SCAF may contain speculative assertions information added by speculation modules that contributed to the resolution of the query. In fact, the query response may contain a set of different options, any of which can be selected by the client (*Response Syntax* in Figure 3). Each of these options may contain multiple speculative assertions that all need to hold true for the analysis result to be sound. The algorithms presented in §3.3 demonstrate how the Orchestrator populates this set of options according to client-selected policies. Note that as opposed to clients, modules within SCAF do not need to be aware of the utilized speculative assertions for a given query.

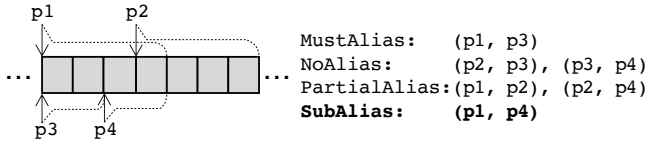



**Figure 4.** Difference between `MustAlias`, `NoAlias`, `PartialAlias`, and `SubAlias`. Arrows represent the pointed memory addresses, and dashed lines denote access sizes. Only the most precise result is presented. Analysis may return `MayAlias` when it cannot infer any other relation.

Each speculative assertion includes (i) a module identifier that specifies which speculation module produced the assertion; (ii) program points that specify where to apply speculation (different for each module); (iii) an estimated cost for validation overhead; and (iv) potential conflict points introduced by the application of this assertion. This information is used by clients to correctly enforce these assertions by applying the required validation code, avoid conflicting options, and consider the cost/benefit of responses. Details about how speculation modules populate this information are presented in a generic fashion in §4.2.1 and on a per-module basis in §4.2.3 and §4.2.4.

Moreover, this paper introduces an additional alias query result: `SubAlias`. This result is returned when a memory location is fully contained within the other memory location of the alias query. `SubAlias` is different from LLVM's `PartialAlias` [37], where two memory objects are known to be overlapping in some way, but one is not necessarily contained within the other. Figure 4 (inspired by [38]) illustrates the differences among alias results.

## 3.3 Orchestrator

The *Orchestrator* coordinates interactions among modules and between modules and the client by forwarding queries

to the modules and by processing query responses (Algorithm 1). It allows modules to remain simple and decoupled, and it can be instantiated with different configurations to accommodate different clients' requirements.

**Algorithm 1:** *handle(query)*

```
Result: Query Response
(module_list, bailout_policy, join_policy) ← getConfig();
final_res ← conservativeResponse(query);
for module in module_list do
    res ← eval(module, query);
    final_res ← join(join_policy, final_res, res);
    if bailout(bailout_policy, final_res) then
        return final_res;
    end
end
return final_res;
```

Modules' implementations only need to respect the query interface, without considering interactions or conflicts with other modules. Clients can easily reconfigure the *Orchestrator* to adjust the received responses without any modification to the modules.

The need for configurability is caused by the presence of speculation modules in SCAF. In traditional memory analysis frameworks [24, 37], the clients are typically indifferent to which module resolved the query; only the result is of interest. However, in SCAF, the same analysis outcome may come with different caveats depending on which modules participated in the resolution of the query. Each speculation module has different requirements in terms of validation for its speculative assertions.

The `join_policy` determines what the *Orchestrator* records for each received response (Algorithm 2). It can either collect all the possible ways a query can be resolved to enable clients to perform global reasoning, or just keep the locally optimal option. Need for global reasoning sources from the fact that a single speculative assertion might be able to resolve with the same cost multiple client's queries as opposed to a cheaper assertion that resolves only one query. The latter is locally better for one particular query, but the former is globally better. Regarding the conflicting results case, it represents an analysis bug if the results are not speculative. If the results are predicated on speculative assertions, it is possible that for different profiling inputs different results appear true. The difference in speculation confidence could determine which one should be preferred.

The `bailout_policy` determines when to stop the search. A default base policy makes the *Orchestrator* immediately return when a definite answer (i.e., the most precise) is found with no attached assertions (i.e., cost-free). Apart from this policy, the *Orchestrator's* search may stop when all the options have been explored (exhaustive search), or when a timeout occurs (clients sensitive to compilation time), or when a definite answer is found regardless of cost, or based on some other heuristic.

**Algorithm 2:** *join(join_policy, r1, r2)*

```
Result: Query Response
/* Define assertion-related semantics                    */
Def O1 + O2 = O1 ∪ O2 ;
Def S1 + S2 = S1 ∪ S2 ;
Def S1 × S2 = {O1 + O2 : O1 ∈ S1, O2 ∈ S2} ;
/* Define order of precision of results                  */
Def pr(NoAlias) == pr(MustAlias) >pr(SubAlias)
>pr(MayAlias);
Def pr(NoModRef) >pr(Mod) == pr(Ref) >pr(ModRef);
(R1, S1) ← r1;
(R2, S2) ← r2;
if pr(R1) >pr(R2) then return r1;
if pr(R1) <pr(R2) then return r2;
/* pr(R1) == pr(R2)                                      */
if R1 == R2 then
    switch join_policy do
        case ALL return (R1, S1 + S2);
        case CHEAPEST return (R1, cheaper(S1, S2));
        case Other Policies ...;
    endsw
end
/* Special Case: Mod and Ref                             */
else if (R1 == Mod and R2 == Ref) or (R1 == Ref and R2 == Mod)
then
    if conflict(S1, S2) then
        return handleConflictingAssertions(r1, r2)
    end
    else
        return (NoModRef, S1 × S2)
    end
end
else
    return handleConflictingResults(r1, r2);
end
```

In our implementation, for simplicity and lack of empirical evidence justifying exposure of all options and exhaustive search, we opt for a greedy search that terminates when a definite result is found and presents only one option to the client.

The *Orchestrator* could also be configured to query any subset of the available modules. For example, a client who wants to avoid speculation can configure the *Orchestrator* to query only memory analysis modules. The ordering of modules is also important as it affects query latency and the effectiveness of greedy approaches. Typically, modules with the smaller average cost of speculative assertions are prioritized. Since memory analysis modules' answers are caveat-free (no validation), they are normally queried first. From among the memory analysis modules, the order could be determined by the query latency.

```
loop L:
    if (rare)
        // no writes to a
        ...
    else
i1:     a = ...;
i2: b = foo(a);
    ...
i3: a = ...;
```

**(a)** Original Code

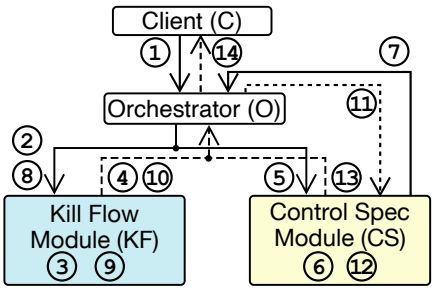
```
loop L:
    // rare path ignored
i1: a = ...;
    // data flow from i3
    // killed by i1
i2: b = foo(a);
    ...
i3: a = ...;
```

**(b)** Speculative View

```
loop L:
    if (rare)
        misspec(branch_tag);
        ...
    else
i1:     a = ...;
i2: b = foo(a);
    ...
i3: a = ...;
```

**(c)** Control Speculation Applied

**Figure 5.** Motivating Code Example




| Step | Flow | Action |
|---|---|---|
| ① | C→O | Call `handle`($\mathbf{q_0}$=`modref`(i3, Before, i2, *L*, *cc*, *dt*, *pdt*)) |
| ② | O→KF | Call `eval`(KF, $\mathbf{q_0}$) |
| ③ | KF | Generate $\mathbf{r_0}$ : (**ModRef**, $\emptyset$) // the flow from i3 to i2 is not killed |
| ④ | KF→O | Return $\mathbf{r_0}$ |
| ⑤ | O→CS | Call `eval`(CS, $\mathbf{q_0}$) |
| ⑥ | CS | Generate premise query $\mathbf{q_1}$: `modref`(i3, Before, i2, *L*, *cc*, ***spec_dt***, ***spec_pdt***) |
| ⑦ | CS→O | Call `handle`($\mathbf{q_1}$) |
| ⑧ | O→KF | Call `eval`(KF, $\mathbf{q_1}$) |
| ⑨ | KF | Generate $\mathbf{r_1}$ : (**NoModRef**, $\emptyset$) // i1 kills the flow from i3 to i2 |
| ⑩ | KF→O | Return $\mathbf{r_1}$ |
| ⑪ | O→CS | Return $\mathbf{r_1}$ |
| ⑫ | CS | Generate control speculation assertion $\mathcal{A}$ (branch never taken), assertion option $O$={$\mathcal{A}$}, and response $\mathbf{r_3}$ : (**NoModRef**, $O$) |
| ⑬ | CS→O | Return $\mathbf{r_3}$ |
| ⑭ | O→C | Return $\mathbf{r_3}$ |

**Figure 6.** A step-by-step example of SCAF. The client wants to determine if there is a cross-iteration data flow from i3 to i2 in the loop in Figure 5a. To that end, it creates a modref query that asks if instruction i3 may read or write the memory footprint of i2 in a later iteration, assuming some static control flow information (*dt*, *pdt*). The kill-flow and control speculation modules synergistically resolve this query, not addressable in isolation by any of these two modules. In Step 9, the kill-flow module perceives the code as in the *Speculative View* in Figure 5b due to the speculative control flow information (*spec_dt*, *spec_pdt*).

### 3.4 SCAF within a Compiler

***SCAF suggests:*** SCAF does *not* perform any transformation. Clients can choose to ignore SCAF's result to avoid paying the cost of its accompanying speculative assertions. SCAF merely makes suggestions. Different configurations of the *Orchestrator* adjust the content of these suggestions, the final decision on what transformations to perform is still left to the client.

***SCAF facilitates planning:*** SCAF avoids the defect of conventional compiler designs where the effect of speculative transformations is only visible after performing the actual transformation. Since SCAF reports the result of queries predicated on speculative assertions, the compiler can perform global reasoning and weigh the impact of applying speculative transforms prior to actually applying them. For example, a parallelization transformation client could query SCAF for all the dependences in a hot loop. Then, to select the set of necessary speculative assertions, this client can formulate an optimization problem considering the removal cost of dependences and the parallelization gains. In the case of multiple clients, a coordinating component could consider the cost/benefit of multiple optimizations simultaneously and prevent redundant speculation validation checks. Finally, rational clients would not apply validation checks for non-leveraged speculative assertions.

The new compiler technology motivated by SCAF is left for future work. This technology includes the re-design of optimizations' interfaces to enable planning and coordination among compiler components. Significant advancements enabled by SCAF are expected for optimizing clients with high performance returns (high enough to tolerate validation and recovery overheads), such as automatic parallelization and vectorization.

### 3.5 Example

This section illustrates concepts and design decisions described in the previous sections with the motivating example from section §2.2.2. For this example (presented again in Figure 5a), a client wants to determine whether there is a cross-iteration dependence from instruction *i*3 to *i*2. Such a client could be a parallelization transformation that needs

to remove all the cross-iteration dependences so that each iteration can execute independently and in parallel.

Edge profiling information allows control speculation to infer that the branch in this code example is never taken (rare condition). We propose that memory analysis and other speculation modules should leverage the full effect of control speculation without performing code transformations. In particular, the speculative assertion that the branch is not taken should be understood by all modules in SCAF as a fact (view misspeculation as impossible) since recovery code, inserted by the client, preserves correctness in the case of misspeculation. The view of the real effect of control speculation is presented in Figure 5b. The speculative view is not intended to show transformed code (SCAF does not change the code), but to explain how the code should be understood by other modules given speculative dominance information. This view of the code enables the kill-flow analysis algorithm to prove that $i1$ kills (overwrites) the cross-iteration data flow from $i3$, and thus disprove the dependence in question.

Figure 6 shows how this code example is handled by SCAF step by step. For simplicity, this example only consists of two modules - a kill-flow memory analysis module and a control speculation module. SCAF enables control speculation to express a speculative control flow of the loop to other modules via a premise query. This premise query is received by the kill-flow module that can resolve the premise query as opposed to the initially received query. In the end, the client receives a `NoModRef` result predicated on the speculative assertion $A$ that the branch is never taken. If the client chooses to leverage the `NoModRef` result and wants to preserve soundness, it would need to insert a function call at the beginning of the taken path to trigger misspeculation, as shown in Figure 5c. For a parallelization transformation client, recovery would involve rollback to the last checkpointed memory state and sequential execution of the original code (without speculation applied) up to the iteration that caused the misspeculation.

In this example, a collaboration between a speculation module and a memory analysis module results in a dependence removal. In general, collaboration in SCAF can also occur among speculation modules or be initiated by memory analysis modules.

Without collaboration, the removed dependence in this example would require memory speculation. Memory speculation just asserts the absence of non-observed during profiling dependences without any understanding of why they are not observed; there is no reasoning. Thus, its validation requires expensive monitoring of the involved operations and comparison of their access patterns. Instead, SCAF manages to inexpensively remove the dependence in question by understanding the reason why this dependence did not manifest during profiling (i.e., not taken branch).

# 4 Implementation

SCAF is implemented on the LLVM Compiler Infrastructure [32] (version 5.0.2). This section describes the memory analysis (§4.1) and speculation (§4.2) modules included in SCAF's implementation.

## 4.1 Memory Analysis Modules

SCAF includes the 13 analysis algorithms described in CAF [24]. Each of these algorithms tries to disprove one of the conditions described in §2.1. In particular, they reason about shape analysis, reachability, flow killing, scalar evolution of pointers, induction variables, and features of the LLVM IR and the C standard library.

Several of these algorithms initiate collaboration by creating premise queries. In CAF, these premise queries can only be resolved by other memory analysis algorithms. In SCAF, both memory analysis and speculation modules attempt to resolve these queries, effectively increasing the impact of the partially resolved queries. Moreover, memory analysis modules can also resolve premise queries generated by speculation modules. An example of collaboration among memory analysis and speculation modules is presented in §3.5.

## 4.2 Speculation Modules

This section describes a design pattern for speculation modules (§4.2.1), enumerates the profilers that guide them (§4.2.2), and finally briefly describes the speculation modules implemented in SCAF (§4.2.3, §4.2.4). For each speculation module, this section describes the effect of its speculative assertions on dependence analysis, the validation code for its assertions, and the possibility of conflicts with other speculation modules.

### 4.2.1 Developing Speculation Modules

To overcome the inherent imprecision of memory analysis algorithms, traditional compiler designs involve speculative transformations. Memory speculation can address the imprecision of memory analysis by asserting the absence of dependences not manifested during profiling. However, memory speculation incurs a high validation cost [7, 16, 55]. To lower the validation cost, state-of-the-art compilers also implement less generic but cheaper-to-validate speculative transformations compared to memory speculation.

In this paper, such a speculative transformation is decomposed into an *analysis* and a *transformation* part. This decomposition exposes the effect of a speculative transformation prior to its application, enabling careful planning. The *analysis* part is a *speculation module* that interprets profile information in terms of dependence analysis, produces speculative assertions, and communicates with the same query language (§3.2) as memory analysis modules. The *transformation* part includes validation code generation that ensures

the correctness of the produced speculative assertions, recovery code generation in case of misspeculation, and runtime support. SCAF's clients apply this transformation part to safely leverage the module's speculative assertions without violating the program's semantics.

***Design of Speculative Assertions:*** Each speculative assertion includes a module identifier, transformation program points, an estimated cost, and conflict points. The identifier is used by clients to identify the corresponding transformation code. The program points specify where to apply the transformation (e.g., a branch instruction for control speculation). The cost enables clients to optimize the selection of applied transformations based on their cost and benefit. Conflict points specify program operations that need to be modified and allow clients to detect ahead of time conflicting transformations (i.e., application of one prevents the application of another).

***Estimated Cost Computation:*** The cost of speculation typically comes from validation and recovery. In this work, we only estimate the validation cost. Validating speculative assertions' adds latency in the common case, not only during recovery. Further, all speculative assertions, in this work, are high-confidence (always hold true during profiling), thus misspeculation is equivalently unlikely for all the assertions. The existence of less conservative speculation schemes with varying misspeculation rates would require modeling of this recovery cost. The total validation cost of a speculative assertion is computed by multiplying a latency estimate of one invocation of the validation code with the execution count (measured during profiling) of the guarded operation. For example, for the case of value prediction on a load operation, the validation code (check that the predicted value matches the loaded value) will execute as many times as the load operation (guarded instruction). The validation cost estimate for one invocation is the average execution time of the validation code observed during profiling runs across several benchmarks and inputs.

***Directives to Minimize Conflicts:*** To minimize conflicts in terms of validation, it is preferable to insert validation code adjacent to speculated operations rather than replacing original program operations. By following this principle, most of the produced speculative assertions in our implementation do not introduce any conflict points. For the rest, orthogonality, in terms of coverage, prevents conflicts.

***Modular Design:*** Each speculation module and its validation code is decoupled and can be developed independently from other modules as long as the module interface described in §3.2 is respected. Development of these speculation modules and integration in SCAF is not more complex than the development of separate speculation transformations as customary in existing research compilers. The main new overhead is additional code to conform to the SCAF's interface. This code, though, is of insignificant complexity compared to the logic for determining the applicability of the transformation or the code for the application of the transformation.

***Design with Collaboration in Mind:*** Our system does better than simply adding speculation modules into the ensemble. Speculation modules in SCAF are designed with collaboration in mind to maximize their impact. Traditionally, speculative techniques are self-contained, resolving dependences in isolation. In SCAF, speculation modules can still directly address dependence queries, but can also generate premise queries, delegated by the *Orchestrator* to memory analysis or other speculation modules. This collaborative environment enables the decomposition of complex speculative techniques to multiple simple speculation modules (e.g., extraction of points-to (§4.2.3), read-only (§4.2.4), short-lived (§4.2.4) modules from separation speculation [25]), and participation of speculation modules in resolution of queries that go beyond their own reasoning (e.g., Figure 6).

#### 4.2.2 Profilers

SCAF's speculation modules use information generated by a set of profilers: (i) an edge profiler that identifies biased branches [32]; (ii) a value-prediction profiler that detects predictable loads [18]; (iii) a pointer-to-object profiler that produces a points-to map, allowing detection of underlying objects for every memory access [25]; and, (iv) an object lifetime profiler that detects short-lived memory objects, namely objects that exist only within a single loop iteration [25].

#### 4.2.3 Base Speculation Modules

The following base speculation modules resolve client queries or premise queries of other modules using profiling information. Base modules do not generate premise queries.

***Pointer-Residue Speculation*** attempts to disambiguate different fields within an object and may also recognize different regular strides across an array. Each pointer is characterized according to the observed during profiling values of its four least-significant bits (residue). This module asserts the absence of dependences between operations with disjoint residue sets (with respect to their access size). Validation of this speculative information is inexpensive, involves bitwise operations that ensure that dynamic pointer values have expected residues, and does not conflict with the validation of other modules' assertions (original code instructions are left unmodified). This speculative technique has been proposed by Johnson [23].

***Points-to Speculation*** identifies underlying objects (allocation sites) for every pointer using a points-to profiler. Using this speculative information, it answers *alias* queries, and may return `SubAlias` (explained in §3.2.3). Validating points-to objects information is, in general, expensive and complicated. Thus, we assign a prohibitively high cost to

points-to assertions that effectively prevents clients from using responses predicated on such assertions. Yet, answers of the points-to module can be leveraged by other speculation modules, such as read-only and short-lived modules (§4.2.4), without paying this high cost. In particular, these modules' validation code separates select memory objects to a separate heap. Since distinguishing objects within a heap is not necessary for these modules, they only need to insert points-to heap checks instead of expensive points-to object checks. In other words, these modules can safely ignore the expensive-to-validate points-to speculation assertion in the premise query response, and replace it with their own assertions.

#### 4.2.4 Factored Speculation Modules

Same as factored memory analysis algorithms in CAF [24], factored speculation modules initiate collaboration by generating premise queries that may be resolved by other speculation or memory analysis modules.

***Control Speculation*** identifies speculatively dead[1] basic blocks using edge profiling. It asserts that speculatively dead instructions cannot source or sink memory dependences. This speculative assertion enables the resolution of client queries and premise queries of other modules.

For example, the control speculation module can address premise queries generated by the reachability algorithms described in CAF [24] (i.e., *Global Malloc, No-Capture Global, No-Capture Source, Unique Access Paths*). These reachability algorithms reason about which object addresses can be stored in particular memory locations. The control speculation module may resolve premise queries related to speculatively unreachable stores to these memory locations, and thus facilitate the resolution of queries related to pointers loaded from these locations.

Additionally, the control speculation module initiates collaboration by generating premise queries that replace static control flow information of received queries with speculative control flow information (in the form of dominator and post-dominator trees). The premise query with the optimistic control flow information is more likely to be resolved by control-flow sensitive analysis modules compared to the original query. If the speculative control flow is proven to be useful by leading to the resolution of a query, control speculation module appends the required speculative control-flow assertions to the query response.

Validation involves the insertion of a function call triggering recovery at the beginning of the speculatively dead path of biased branches. This validation does not modify original code instructions. It thus does not introduce conflicts with other speculative assertions, as opposed to other schemes (e.g., [41]) that propose the replacement of biased branches with assertions. The validation cost of control speculation is practically zero since the biased branch is computed anyway. The only potential overhead is the cost of recovery in the unlikely case of misspeculation.

***Value Prediction*** identifies predictable loads using profiling information. It resolves data dependences that sink into or source from these predictable loads. The value prediction module can also interpret predictable loads as kill operations to resolve additional queries leveraging the *no-kill* condition from §2.1. If a predictable load post-dominates the source of a queried dependence and dominates the destination, the value prediction module generates premise queries to compare the memory footprint of the predictable load with the footprint of the dependent instructions. Must-alias result for either of the two premise queries enables the value prediction module to assert a lack of dependence. Validation is inexpensive, involves a simple comparison of the loaded value with the predicted one, and it does not conflict with other assertions.

***Read-only*** identifies memory objects that are never written to within a target loop based on profiling information [25]. This module generates premise queries to compare the memory locations of read-only objects with the memory locations involved in received queries. It asserts that read-only memory locations cannot be written to and asserts disjointedness of read-only objects from pointers to other objects. Johnson et al. [25] have shown that validation of these assertions is inexpensive via separation of read-only memory objects to a separate heap and simple bitwise operations on computed pointers to check points-to heap assertions. Note that in this work, we separate and decompose the analysis part of separation speculation [25] to simple modules that collaboratively infer at least the same properties as the monolithic design proposed in [25]. Further, this work avoids points-to heap checks if the premise query reports `MustAlias` with zero cost. Since read-only assertions require re-allocation of the involved memory objects to the read-only heap, they conflict with any other assertions that require modification of the allocation sites of the same memory objects.

***Short-lived*** identifies memory objects that only exist within one iteration of the loop of interest using profiling information [25, 28, 55]. Similarly to the read-only module, it generates premise queries to compare the memory locations involved in the original query with the locations of short-lived objects. It asserts the absence of cross-iteration dependences on any access to short-lived objects and asserts disjointedness of these objects from pointers to other objects. Similarly to the read-only module's validation, validation of the short-lived module's assertions is inexpensive and introduces conflicts on the allocation sites of the involved short-lived objects. Note that the short-lived and the read-only objects are disjoint sets, and thus no conflict between

[1] We focus on high-confidence speculation and thus only never executed during profiling basic blocks are considered.

their assertions is possible. In addition, the short-lived module's assertions additionally require a simple check at the end of every loop iteration that verifies that the count of allocated short-lived objects equals the count of freed ones.

#### 4.2.5 Recovery

Clients utilizing SCAF's query responses with speculative assertions need to insert the corresponding validation code (described in §4.2.3, §4.2.4) to preserve the semantics of the original code. At runtime, if the validation checks fail, misspeculation occurs and recovery code should be activated. Therefore, clients that leverage speculative assertions should support recovery and separation of speculative and non-speculative state. There is a rich literature of recovery mechanisms for systems that speculate memory dependences. These mechanisms, that SCAF's clients can leverage, are summarized in two main categories: process-based [13, 25, 27, 29, 46] and thread-based [22, 40, 54–56] schemes.

## 5 Evaluation

***Benchmark Selection:*** We evaluate SCAF against 16 C/C++ benchmarks from the SPEC suites (SPEC CPU 92/95/2000/2006/2017) [51]. We exclude Fortran SPEC benchmarks due to lack of Fortran front-end support (Flang [17] not supported in LLVM 5.0). We also exclude other C/C++ SPEC benchmarks due to the limitations of our profilers' implementation. All profilers (§4.2.2) except for the edge profiler (LLVM version) are implemented in-house, lacking industrial-level robustness in implementation. Benchmarks for which at least one profiler failed to produce results were rejected since only a subset of the speculation modules would be applicable. Problems include unanticipated code patterns that break code instrumentation, runtime errors of instrumented executables, and prohibitively large profile data.

***Hot Loops:*** SCAF is evaluated on the hot loops of the evaluated benchmarks. These are the loops that comprise at least 10% of total program execution time and iterate at least 50 times on average per invocation. We evaluate on hot loops because improvements in memory dependence analysis for hot loops are expected to be more beneficial to clients compared to other parts of the benchmarks.

***Profiling Data:*** We gather profiling information using the *train* inputs from the SPEC benchmark suites.

***Client:*** We evaluate SCAF with a Program Dependence Graph (PDG) client [15]. For each hot loop, the PDG client performs an intra-iteration and a cross-iteration dependence query for each pair of memory operations (each dependence is valued equally). Quantifying the impact of SCAF at the optimizing client level is not possible given today's compiler technology (see §3.4) and is left for future work.

***Metric:*** Same as in prior work [24], we utilize the `%NoDep` metric as a measure of analysis precision. This metric denotes the percent of dependence queries for which the evaluated analysis framework reports no flow, anti, or output dependence. The coverage in terms of dependence removal is a direct measure of SCAF's impact, as opposed to the performance response that is tied to the specifics of the evaluated optimizing client and thus an indirect measure. While the performance impact for an optimizing client is not measured, the selected metric is indicative of such an impact. Performance is highly correlated with the cost of memory dependence removal in hot loops for certain types of clients, such as parallelization techniques [16, 40, 55].

***Best Prior Approach:*** We evaluate the positive effect of collaboration among speculation modules and between memory analysis and speculation modules by comparing SCAF against the best prior approach that integrates speculation into dependence analysis: *composition by confluence* (§2.2.1). This approach resembles prior proposals [28, 40, 57, 61] that utilize speculative techniques independently, each handling memory dependences on its own without interactions with other speculation or memory analysis modules. In *composition by confluence*, each dependence query is passed to each module in isolation, and the confluence of individual results is returned. To avoid taking credit for contributions of prior work (CAF [24] supports collaboration among memory analysis modules), we treat all the memory analysis modules as one component within which collaboration is permitted. We refer to this component as CAF. Both composition by *collaboration* (SCAF) and by *confluence* use the same memory analysis and speculation modules. Same as SCAF, *composition by confluence* does not use memory speculation, and only reports cheap-to-validate assertions (responses that include points-to speculation assertions are discarded).

***Memory Speculation:*** Memory speculation is the most commonly used and applicable speculation technique [25, 28, 40, 47, 56]. It asserts the absence of non-observed memory dependences using a loop-sensitive memory dependence profiler [8]. Yet, memory speculation is the most expensive speculation technique. Excessive usage of memory speculation often negates its enabling effect [7, 16, 55]. To validate that a memory dependence between two operations is not manifested at runtime, the access pattern of these two memory operations needs to be monitored at runtime. A shadow memory is commonly used to keep track of accessed memory locations for all the speculative accesses [25, 43, 47]. This is expensive for software-only systems where monitoring of large read and write sets results in dramatic slowdowns [25, 46]. Figure 7b shows an assembly code snippet of a typical memory speculation validation code and compares it with an example of a cheap-to-validate speculation from SCAF (points-to heap check in Figure 7a). Validation of the rest of speculation modules in SCAF (§4.2) is not more

```
 r0 := addr

point_to_heap_check:
 r1 = r0 & MASK
 br r1 != EXPECTED,misspec
```

**(a)** Inexpensive

```
 r0 := addr
 r1 := type

mem_spec_check:
 r2 = r0 | SHADOW_MASK
 r3 = M[r2]
 r4 = check_meta(r3, r1)
 br r4 == FAIL, misspec
 r5 = update_meta(r3, r1)
 M[r2] = r5
```

**(b)** Expensive

**Figure 7.** Speculation validation code examples. Validation of SCAF's modules involves only a few bitwise/arithemetic/branch instructions, while the memory speculation check involves many more operations, including memory accesses.

complicated than the simple check in Figure 7a. Note that if the client is a parallelization transformation and the speculated dependence is a cross-iteration one, memory speculation validation additionally involves the communication of memory footprints among parallel workers. In contrast, SCAF only employs assertions with inexpensive checks that, even for parallel execution, are performed locally by each worker. Overall, SCAF aims to reduce the gap between the dependence coverage of memory speculation and cheap-to-validate speculation and enable more profitable speculative optimizations.

### 5.1 Benefit of Collaboration

Figure 8 compares SCAF, *composition by confluence*, memory speculation, and CAF [24] using the `%NoDep` metric for the PDG client. Since resolving a memory dependence within a frequently executed loop has typically a larger impact to that within a less frequently executed loop, we record `%NoDep` at a loop granularity and weight it by the loop's execution time. `%NoDep` for each benchmark is, therefore, a weighted sum of `%NoDep` per loop.

SCAF increases, on average, the dependence coverage by 68.35% (56.27% for geomean) compared to *composition by confluence*. Note that SCAF outperforms *composition by confluence* for all the evaluated benchmarks; in some cases, the improvement is too small to observe in the graph. Given that both SCAF and *composition by confluence* use the same inexpensive-to-validate speculative assertions, the coverage improvement highlights how SCAF maximizes the impact of these speculative assertions by exposing them to all the modules in the framework.

By maximizing the impact of inexpensive speculative assertions, SCAF effectively reduces the need for expensive-to-validate memory speculation for dependence removal. In fact, Figure 8 shows a dramatic reduction of the memory speculation bar (58.41% geomean). This reduction means that SCAF removes with cheap-to-validate speculation dependences for which prior work would require memory speculation. Moreover, memory speculation asserts the absence of individual dependences, while a cheap assertion, such as control speculation assertion, may resolve (either in isolation or collaboratively) multiple dependences. In other words, to achieve the same dependence coverage as prior work, SCAF uses not only cheaper assertions but also fewer. Therefore, these results strongly indicate that SCAF decreases validation costs compared to the best prior approach, despite the lack of optimizing client results.

Figure 9 compares SCAF with *composition by confluence* in terms of the `%NoDep` metric of the PDG client for each of the hot loops within the evaluated SPEC benchmarks. SCAF outperforms *composition by confluence* for 37 out of 56 hot loops from the evaluated SPEC benchmarks. For these loops, collaboration enables removal of dependences non-addressable by any module in isolation. For the rest of the loops, both schemes have the same precision. Lack of benefit by SCAF on the latter loops is mostly due to high coverage of non-observed dependences by *composition by confluence*, leaving few (if any) opportunities for increasing the impact of cheap speculation. These loops are mainly found in `056.ear`, `129.compress`, `164.gzip`, `179.art` benchmarks.

### 5.2 Contributions of Modules to Collaboration

This section evaluates which modules within SCAF participate in collaborations across the 16 evaluated benchmarks, and thus contribute to the improvements in the `%NoDep` metric of the PDG client (discussed in §5.1). Collaboration

**Table 2.** Collaboration coverage of modules in SCAF on the benchmark, loop, and improved query (i.e., query benefited by collaboration) levels. The percentage of a module denotes the coverage of beneficial collaboration involving the module for the population of a certain level (e.g., the 93.75% coverage of CAF on the benchmark level means that CAF is used in collaboration with other modules for 93.75% of benchmarks for removal of dependences unresolvable with *composition by confluence*).

| Analysis Modules | | Collaboration Coverage (%) | | |
|---|---|---|---|---|
| | | Benchmark Level | Loop Level | Improved Query Level |
| Memory Analysis (CAF) | | 93.75 | 42.86 | 40.02 |
| Spec. Modules | Read-only | 87.50 | 53.57 | 71.52 |
| | Value Prediction | 12.50 | 3.57 | 0.11 |
| | Pointer-Residue | 6.25 | 1.79 | 0.00 |
| | Control Speculation | 75.00 | 30.36 | 18.57 |
| | Points-to | 87.50 | 53.57 | 81.32 |
| | Short-lived | 6.25 | 1.79 | 9.80 |
| Among Speculation Modules | | 87.50 | 53.57 | 81.32 |
| Between CAF and Speculation | | 93.75 | 42.86 | 40.02 |
| All | | 100.00 | 66.07 | 100.00 |

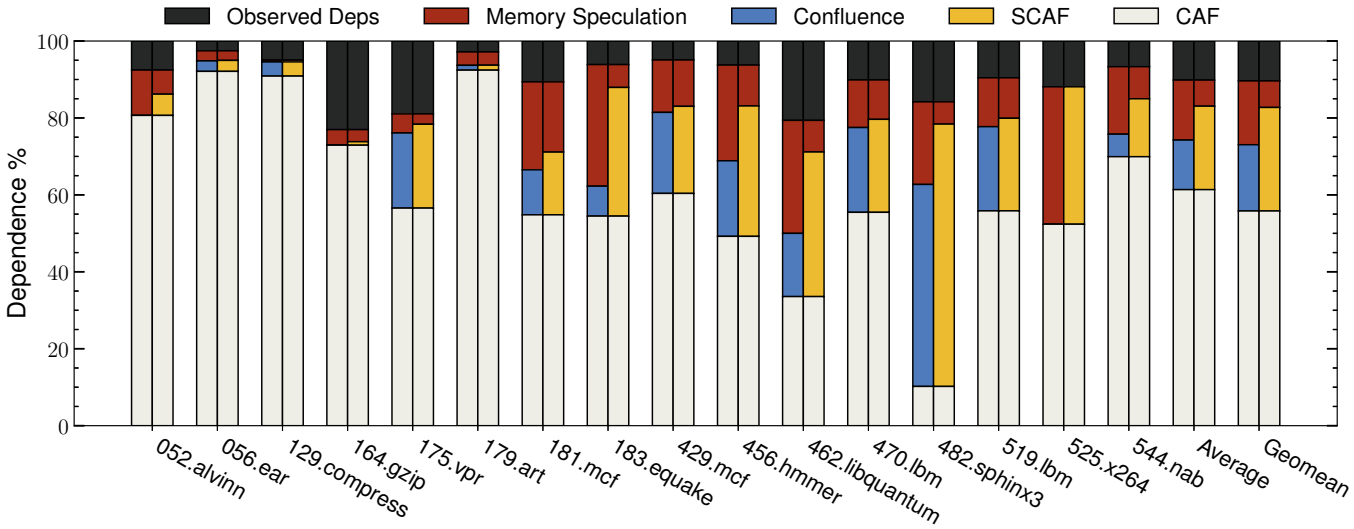



**Figure 8.** Dependences coverage by different schemes. *CAF* denotes dependences disproven by memory analysis (CAF [24]). *Confluence* and *SCAF* show additional dependences removed using inexpensive speculation without and with collaboration, respectively. *Memory speculation* asserts the absence of the remaining dependences that do not manifest during profiling. *Observed deps* are dependences that manifest during profiling.

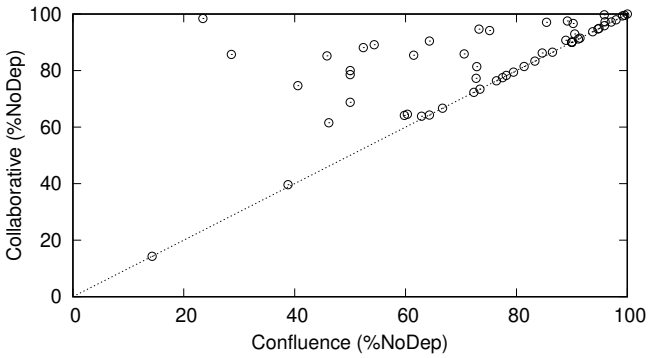



**Figure 9.** *Composition by Collaboration* (SCAF) compared with *Composition by Confluence*. Each point is a hot loop. Collaboration performs better on loops above the diagonal.

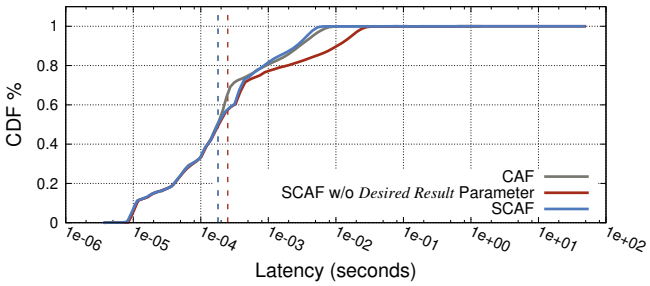



**Figure 10.** CDF of query latency for CAF [24], SCAF without *Desired Result* parameter, and SCAF. The vertical colored dashed lines represent the geomean of each. The geomeans of SCAF and CAF are overlapping.

exhibits when two or more modules achieve higher precision than the confluence of their individual results.

Table 2 presents each module's contribution to collaboration. We treat all the memory analysis modules as one single component (CAF) and focus on the interactions of memory analysis as a whole with speculation modules.

These results *strongly* corroborate the hypothesis that collaboration between memory analysis and speculation modules is beneficial. Memory analysis modules collaborate with at least one speculation module for 15 out of 16 evaluated benchmarks, for 42.86% of the evaluated hot loops, and for 40.02% of benefited from collaboration queries.

Notice also that the control speculation module participates in numerous fruitful collaborations, indicating the usefulness of providing speculative control flow information to other modules. The rest of speculation modules also profitably collaborate in varying degrees.

Furthermore, these results show that more than two components contribute to the resolution of certain queries because the sum of the percentages of all the analysis modules for queries benefited by collaboration is bigger than 200%.

### 5.3 Query Latency

Figure 10 presents the cumulative distribution function (CDF) of the query latency for CAF [24], SCAF without *Desired Result* parameter, and SCAF. All the queries performed by the PDG client are considered. Time is measured in processor cycles on a 14-core Intel Xeon CPU E5-2697 v3 processor running at 2.60GHz (turbo-boost disabled) with 768GB of available memory. SCAF's query latency is reduced by 27.50%

(geomean) with the introduction of the *Desired Result* parameter. Compared with CAF, SCAF, despite using more analysis modules (i.e., addition of speculation modules), increases the geomean query latency by only 1.61%. Finally, 95% of queries are serviced by SCAF within 2.6ms.

## 6 Related Work

Johnson [23] proposes a design integrating speculation in a collaborative analysis framework (CAF [24]); however, there has been no published work implementing this design. Additionally, merely adding existing speculative techniques into an analysis ensemble is not sufficient to enable collaboration. Speculative techniques need to be re-designed with collaboration in mind. Traditionally, each speculative technique is self-contained. Instead, speculation modules in SCAF extend their impact by initiating collaboration and requesting assistance from other modules. SCAF also decomposes complex and monolithic speculative techniques mentioned in [23] to simple analysis modules. Moreover, the query language used in CAF is insufficient to fully leverage speculative information, most prominently control flow information. SCAF's query language supports the communication of both data and control flow information among modules. Finally, Johnson's proposal is tied to a particular client [26], while SCAF is client-agnostic. SCAF specifies the required speculative assertions along with each query answer, allowing clients to decide on how to act upon this information.

Other works [2, 12, 14] also explored integrating speculative information into static analysis, but in a monolithic fashion. Static analysis algorithms in these prior works are tightly coupled with specific speculative information. By contrast, SCAF is a modular and thus easily extensible framework in which a broad set of memory analysis and speculation modules synergistically resolve queries while being fully decoupled.

In terms of static analysis, prior works [5, 6, 11, 24, 30, 34, 42] explore collaboration among analysis algorithms. However, these works do not leverage speculation and are thus restrained by the inherent imprecision of memory analysis.

To overcome the imprecision of memory analysis, hybrid analysis [49] and sensitivity analysis [48] explore the combination of static and run-time analysis. Static analysis is used to extract run-time checks, which determine if the parallelized code is safe to execute. However, unlike profile-driven approaches, run-time analysis offers limited coverage and small improvement over memory analysis. SCAF instead uses profiling to exploit commonly executed patterns and avoid arbitrarily complex run-time checks.

Several speculative automatic parallelization works [28, 57] employ a *composition by confluence* approach where they first produce a conservative PDG using memory analysis and successively refine it using a series of speculative techniques. This approach does not allow interactions among speculative techniques and memory analysis algorithms. SCAF allows parallelization clients to identify more parallelizable regions due to higher precision achieved via collaboration.

Other works combine profile-driven approaches with memory analysis for clients beyond the scope of parallelization. Lin et al. [36] propose a speculative single static assignment (SSA) form that incorporates memory and control speculation. However, memory analysis does not leverage speculative information, and only low-level optimizations are targeted. Manilov et al. [39] use memory analysis enhanced with profiling information to recognize iterators of loops. However, the authors rely on profile-guided data flow information that would be expensive to validate. SCAF reduces validation overheads for clients by utilizing various types of cheap-to-validate speculation techniques and achieves high precision by enabling the collaboration of analysis modules.

## 7 Conclusion

This paper presents the design, implementation, and evaluation of SCAF, a modular and collaborative dependence analysis framework that computes the full impact of speculation on memory dependence analysis. In SCAF, speculation modules and memory analysis modules with independent implementations work together to resolve memory dependence queries. SCAF enables judicious use of speculation to address memory dependences that would otherwise limit optimizations or lead to expensive-to-validate memory speculation. Relative to the best prior speculation-aware dependence analysis technique, SCAF dramatically reduces the need for expensive-to-validate memory speculation in the hot loops of all 16 evaluated C/C++ SPEC benchmarks. Given these results, we believe that SCAF is beneficial for memory analysis sensitive clients, and a necessary step toward robust automatic parallelization.

## Acknowledgments

We thank the Liberty Research Group for their support and feedback during this work. We also thank Alexandra Jimborean and the anonymous reviewers for their insightful comments and suggestions. This work was supported by the National Science Foundation (NSF) through Grants CCF-1814654 and CNS-1763743. All opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

## References

[1] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Technical Report.

[2] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I. August. 2020. Perspective: A Sensible Approach to Speculative Automatic Parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages*

*and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, Lausanne, Switzerland, 351–367. https://doi.org/10.1145/3373376.3378458

[3] Utpal Banerjee. 1994. *Loop Parallelization*. Springer US. https://doi.org/10.1007/978-1-4757-5676-0

[4] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: precise refutations for heap reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, Seattle, Washington, USA, 275–286. https://doi.org/10.1145/2491956.2462186

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception analysis and points-to analysis: better together. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09)*. Association for Computing Machinery, Chicago, IL, USA, 1–12. https://doi.org/10.1145/1572272.1572274

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. 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)*. Association for Computing Machinery, Orlando, Florida, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[7] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy? *Queue* 6, 5 (Sept. 2008), 46–58. https://doi.org/10.1145/1454456.1454466

[8] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. 2004. Data Dependence Profiling for Speculative Optimizations. In *Compiler Construction (Lecture Notes in Computer Science)*, Evelyn Duesterwald (Ed.). Springer, Berlin, Heidelberg, 57–72. https://doi.org/10.1007/978-3-540-24723-4_5

[9] William Y. Chen, Scott A. Mahlke, and Wen-mei W. Hwu. 1992. Tolerating First Level Memory Access Latency in High-Performance Systems. In *Proceedings of the 1992 International Conference on Parallel Processing*. 37–43.

[10] M. Cintra and J. Torrellas. 2002. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*. 43–54. https://doi.org/10.1109/HPCA.2002.995697 ISSN: 1530-0897.

[11] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. 2011. The Reduced Product of Abstract Domains and the Combination of Decision Procedures. In *Foundations of Software Science and Computational Structures (Lecture Notes in Computer Science)*, Martin Hofmann (Ed.). Springer, Berlin, Heidelberg, 456–472. https://doi.org/10.1007/978-3-642-19805-2_31

[12] David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2018. Optimistic Hybrid Analysis: Accelerating Dynamic Analysis through Predicated Static Analysis. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Association for Computing Machinery, Williamsburg, VA, USA, 348–362. https://doi.org/10.1145/3173162.3177153

[13] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. 2007. Software behavior oriented parallelization. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, San Diego, California, USA, 223–234. https://doi.org/10.1145/1250734.1250760

[14] Manel Fernández and Roger Espasa. 2002. Speculative Alias Analysis for Executable Code. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*. IEEE Computer Society, Washington, DC, USA, 222–231.

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[16] Jordan Fix, Nayana P. Nagendra, Sotiris Apostolakis, Hansen Zhang, Sophie Qiu, and David I. August. 2018. Hardware Multithreaded Transactions. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Association for Computing Machinery, Williamsburg, VA, USA, 15–29. https://doi.org/10.1145/3173162.3173172

[17] Flang Project. 2019. Flang: a Fortran Compiler Targeting LLVM. https://github.com/flang-compiler/flang.

[18] Freddy Gabbay and Avi Mendelson. 1997. Can program profiling support value prediction?. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30)*. IEEE Computer Society, Research Triangle Park, North Carolina, USA, 270–280.

[19] Rakesh Ghiya and Laurie J. Hendren. 1996. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*. Association for Computing Machinery, St. Petersburg Beach, Florida, USA, 1–15. https://doi.org/10.1145/237721.237724

[20] Bolei Guo, Neil Vachharajani, and David I. August. 2007. Shape analysis with inductive recursion synthesis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, San Diego, California, USA, 256–265. https://doi.org/10.1145/1250734.1250764

[21] Michael Hind. 2001. Pointer analysis: haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01)*. Association for Computing Machinery, Snowbird, Utah, USA, 54–61. https://doi.org/10.1145/379605.379665

[22] Jialu Huang, Prakash Prabhu, Thomas B. Jablin, Soumyadeep Ghosh, Sotiris Apostolakis, Jae W. Lee, and David I. August. 2016. Speculatively Exploiting Cross-Invocation Parallelism. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. Association for Computing Machinery, Haifa, Israel, 207–221. https://doi.org/10.1145/2967938.2967959

[23] Nick P Johnson. 2015. *Static Dependence Analysis in an Infrastructure for Automatic Parallelization*. PhD Thesis. Department of Computer Science, Princeton University, Princeton, NJ, United States.

[24] Nick P. Johnson, Jordan Fix, Stephen R. Beard, Taewook Oh, Thomas B. Jablin, and David I. August. 2017. A collaborative dependence analysis framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Austin, USA, 148–159. https://doi.org/10.1109/CGO.2017.7863736

[25] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. 2012. Speculative separation for privatization and reductions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. Association for Computing Machinery, Beijing, China, 359–370. https://doi.org/10.1145/2254064.2254107

[26] Nick P. Johnson, Taewook Oh, Ayal Zaks, and David I. August. 2013. Fast condensation of the program dependence graph. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, Seattle, Washington, USA, 39–50. https://doi.org/10.1145/2491956.2491960

[27] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. 2009. Fast Track: A Software System for Speculative Program Optimization. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. IEEE Computer Society, USA, 157–168. https://doi.org/10.1109/CGO.2009.18

[28] Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. 2012. Automatic speculative DOALL for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. Association for Computing Machinery, San Jose, California, 94–103. https://doi.org/10.1145/2259016.2259029

[29] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. 2010. Scalable Speculative Parallelization on Commodity Clusters. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, USA, 3–14. https://doi.org/10.1109/MICRO.2010.19

[30] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. 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)*. Association for Computing Machinery, Baltimore, Maryland, 1–12. https://doi.org/10.1145/1065167.1065169

[31] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems* 1, 4 (Dec. 1992), 323–337. https://doi.org/10.1145/161494.161501

[32] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04)*. IEEE Computer Society, Palo Alto, California, 75.

[33] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, San Diego, California, USA, 278–289. https://doi.org/10.1145/1250734.1250766

[34] Ondrej Lhotak. 2006. *Program analysis using binary decision diagrams*. phd. McGill University, CAN. ISBN-13: 9780494251959.

[35] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology* 18, 1 (Oct. 2008), 3:1–3:53. https://doi.org/10.1145/1391984.1391987

[36] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. 2003. A compiler framework for speculative analysis and optimizations. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03)*. Association for Computing Machinery, San Diego, California, USA, 289–299. https://doi.org/10.1145/781131.781164

[37] LLVM Project. 2019. LLVM Alias Analysis Infrastructure. http://llvm.org/docs/AliasAnalysis.html.

[38] Maroua Maalej and Laure Gonnord. 2015. *Do we still need new Alias Analyses?* report. Université Lyon Claude Bernard / Laboratoire d'Informatique du ParallÃ¨lisme. https://hal.inria.fr/hal-01228581

[39] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. 2018. Generalized profile-guided iterator recognition. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. Association for Computing Machinery, Vienna, Austria, 185–195. https://doi.org/10.1145/3178372.3179511

[40] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, Dublin, Ireland, 166–176. https://doi.org/10.1145/1542476.1542495

[41] Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. 2007. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07)*. Association for Computing Machinery, San Diego, California, USA, 174–185. https://doi.org/10.1145/1250662.1250684

[42] Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems* 1, 2 (Oct. 1979), 245–257. https://doi.org/10.1145/357073.357079

[43] Taewook Oh, Stephen R. Beard, Nick P. Johnson, Sergiy Popovych, and David I. August. 2017. A Generalized Framework for Automatic Scripting Language Parallelization. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 356–369. https://doi.org/10.1109/PACT.2017.28

[44] Manohar K. Prabhu and Kunle Olukotun. 2003. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)*. Association for Computing Machinery, San Diego, California, USA, 1–12. https://doi.org/10.1145/781498.781500

[45] William Pugh. 1991. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing (Supercomputing '91)*. Association for Computing Machinery, Albuquerque, New Mexico, USA, 4–13. https://doi.org/10.1145/125826.125848

[46] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. 2010. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems (ASPLOS XV)*. Association for Computing Machinery, Pittsburgh, Pennsylvania, USA, 65–76. https://doi.org/10.1145/1736020.1736030

[47] Lawrence Rauchwerger and David Padua. 1995. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *ACM SIGPLAN Notices* 30, 6 (June 1995), 218–232. https://doi.org/10.1145/223428.207148

[48] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. 2007. Sensitivity analysis for automatic parallelization on multi-cores. In *Proceedings of the 21st annual international conference on Supercomputing (ICS '07)*. Association for Computing Machinery, Seattle, Washington, 263–273. https://doi.org/10.1145/1274971.1275008

[49] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2003. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. *International Journal of Parallel Programming* 31, 4 (Aug. 2003), 251–283. https://doi.org/10.1023/A:1024597010150

[50] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1996. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*. Association for Computing Machinery, St. Petersburg Beach, Florida, USA, 16–31. https://doi.org/10.1145/237721.237725

[51] spec [n.d.]. Standard Performance Evaluation Corporation. http://www.spec.org.

[52] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*. Association for Computing Machinery, St. Petersburg Beach, Florida, USA, 32–41. https://doi.org/10.1145/237721.237727

[53] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry. 2002. Improving value communication for thread-level speculation. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*. 65–75. https://doi.org/10.1109/HPCA.2002.995699 ISSN: 1530-0897.

[54] Chen Tian, Min Feng, and Rajiv Gupta. 2010. Speculative parallelization using state separation and multiple value prediction. In *Proceedings of the 2010 international symposium on Memory management (ISMM '10)*. Association for Computing Machinery, Toronto, Ontario, Canada, 63–72. https://doi.org/10.1145/1806651.1806663

[55] Chen Tian, Min Feng, and Rajiv Gupta. 2010. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. Association for Computing Machinery, Toronto, Ontario, Canada, 62–73. https://doi.org/10.1145/1806596.1806604

[56] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. 2008. Copy or Discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, USA, 330–341. https://doi.org/10.1109/MICRO.2008.4771802

[57] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative Decoupled Software Pipelining. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 49–59. https://doi.org/10.1109/PACT.2007.4336199 ISSN: 1089-795X.

[58] Steven Wallace, Brad Calder, and Dean M. Tullsen. 1998. Threaded multiple path execution. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*. IEEE Computer Society, Washington, DC, USA, 238–249.

[59] John Whaley and Monica S. Lam. 2004. 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 '04)*. Association for Computing Machinery, Washington DC, USA, 131–144. https://doi.org/10.1145/996841.996859

[60] Qiang Wu, Artem Pyatakov, Alexey Spiridonov, Easwaran Raman, Douglas W. Clark, and David I. August. 2004. Exposing Memory Access Regularities Using Object-Relative Memory Profiling. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04)*. IEEE Computer Society, Palo Alto, California, 315.

[61] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. 2008. Uncovering hidden loop level parallelism in sequential applications. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 290–301. https://doi.org/10.1109/HPCA.2008.4658647 ISSN: 2378-203X.