A Sensible Approach to Speculative Automatic Parallelization

Sotiris Apostolakis

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: David I. August

January 2021

Abstract

The promise of automatic parallelization, freeing programmers from the error-prone and time-consuming process of making efficient use of parallel processing resources, remains unrealized. For decades, the imprecision of memory analysis limited the applicability of automatic parallelization. The introduction of speculation to automatic parallelization overcame these applicability limitations but caused profitability problems due to high communication and bookkeeping costs for speculation validation and commit.

This dissertation shifts the focus from applicability to profitability by making speculative automatic parallelization more efficient. Unlike current approaches that perform analysis and transformations independently and in sequence, the proposed system integrates, without loss of modularity, memory analysis, speculative techniques, and enabling transformations.

Specifically, this dissertation involves three main contributions. First, it introduces a novel speculation-aware collaborative analysis framework (SCAF) that minimizes the need for high-overhead speculation. Second, it proposes a new parallelizing-compiler design that enables careful planning. Third, it presents new efficient speculative privatization transformations that avoid privatization overheads of prior work.

SCAF learns of available speculative information via profiling, computes its 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, SCAF dramatically reduces the need for expensive-to-validate memory speculation in the hot loops of all 16 evaluated C/C++ SPEC benchmarks.

The introduction of planning in the compiler enables the selection of parallelization-enabling transformations based on their cost and overall impacts. The benefit is twofold. First, the compiler only applies a minimal-cost set of enabling transformations. Second,

this decision-making process makes the addition of new efficient speculative enablers possible, including the proposed privatization transformations.

This dissertation integrates these contributions into a fully-automatic speculative-DOALL parallelization framework for commodity hardware. By reducing speculative parallelization overheads in ways not possible with prior parallelization systems, this work obtains higher overall program speedup (23.0× for 12 general-purpose C/C++ programs running on a 28-core shared-memory machine) than Privateer (11.5×), the prior automatic speculative-DOALL system with the highest applicability.

Acknowledgments

First and foremost, I would like to thank my advisor Prof. David I. August for his support and guidance over the years. I am grateful for his faith in me and his constant encouragement during challenging times. He taught me a great deal about research. He taught me to set ambitious and impactful goals and keep a positive and 'can be done' attitude. Moreover, he helped me improve my writing and presentation skills by showing me how to frame my research clearly and convincingly. I also appreciate that he gave me the freedom to pursue my research interests and ideas. Finally, the culture of collaboration and solidarity he has cultivated in the Liberty Research Group made my research much more enjoyable and rewarding.

I thank the rest of my dissertation committee: Prof. Andrew Appel, Prof. Brian Kernighan, Prof. Simone Campanoni, and Prof. Zachary Kincaid. I want to additionally thank Prof. Campanoni and Prof. Kincaid for taking the time to serve as readers on my committee. Their feedback helped improve the quality of this dissertation. Prof. Campanoni has also been a great collaborator for the work presented in this dissertation.

I thank the members of the Liberty Research Group for all their help throughout the years. I thank Stephen, Jordan, Deep, Heejin, Nayana, and Hansen for welcoming me to Princeton and to the group. I want to especially thank Hansen, who has been a great friend and collaborator. Further, I want to thank the newest members of the group that brought new energy and enthusiasm: Ziyang, Greg, Bhargav, Zujun, and Ishita. I want to especially thank Ziyang, Greg, and Zujun for their interest in my research and their contributions to my dissertation work over the last two years. Finally, I want to thank Nick and Taewook, who are alumni of the group. Nick's work laid the groundwork for my research, and it was an invaluable source of inspiration. Taewook was an excellent mentor during my summer internship at Facebook.

I thank the administrative staff of the Department of Computer Science at Princeton. In particular, I thank Nicki Gotsis for helping me with all sorts of bureaucratic matters and

Pamela DelOrefice for taking care of travel grants and reimbursements. I also thank the Davis International Center staff for their help on all immigration-related issues.

I am deeply grateful to all my friends for everything we shared during the past five years. I thank the many friends I was lucky to meet at Princeton that helped me adapt to a new way of life both academically and socially. I also thank my long-time friends from back home in Greece. Our summer and Christmas trips helped me recharge my batteries. Thank you for your enduring support. The Greek community at Princeton was also a great support group. Our nights at Fine Hall, dinners, and various events helped me blow off some steam. I want to especially thank Nick and Themis for all the fun moments we shared and for patiently hearing me complain about work at times. I thank Maria for her patience, encouragement, and love. I thank her for always being there for me, and I cherish all the moments we spent together.

I am thankful to my parents for their unconditional love and support, for their personal sacrifices to provide me the education and opportunities that enabled my path, and for the values they instilled in me. I also thank my brother for always believing in me.

I gratefully acknowledge generous financial support by the Siebel Scholars Foundation through the Siebel Scholar award, the Seeger Center for Hellenic Studies through the Stanley J. Seeger fellowship, and the National Science Foundation (NSF) through grants CCF-1814654, CNS-1441650, and CCF-1439085.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Using PThreads [91], Map-Reduce [24], OpenMP [70], and other libraries and languages, programmers routinely produce coarse-grained parallel programs even at the warehouse scale. Such programs are not ideally suited for multicore as they tend to stress multicore's shared resources, such as caches and memory bandwidth. Manually extracting parallelism fine-grained enough for multicore remains a challenge despite developments in parallel programming languages, parallel libraries, and tools [102, 69].

At the other end of the parallelism granularity spectrum, compilers and out-of-order processors consistently extract instruction-level parallelism (ILP) from programs without any programmer intervention. Unfortunately, despite progress in recent years, automatic parallelization is not yet a reliable solution for the extraction of multicore-appropriate parallelism.

Parallelizing compilers integrate program analysis, enabling transformations, and parallelization patterns to find work that can execute concurrently. Automatic parallelization naturally focuses on loops because that is where programs spend their time. An essential aspect of program analysis in a parallelizing compiler is memory analysis because the compiler must understand memory access patterns to divide work across threads or processes. Enabling transformations use control flow and data flow facts from analysis to make the

code amenable to a given parallelization pattern. Examples of enabling transformations include: i) loop skewing, which re-arranges array accesses to move cross-iteration dependences out of inner loops; ii) reduction, which expands storage locations to relax ordering constraints on associative and commutative operations; and iii) privatization, which creates private data copies for each worker process to remove contention caused by the reuse of data structures. Many parallelization patterns exist, but the most desirable is DOALL, the independent execution of loop iterations.

For decades, parallelizing compilers only performed enabling transformations that could be proven correct with respect to the facts provided by static analyses [10, 100, 21, 71, 78, 15]. While this approach showed some success in scientific codes, its reliance on memory analysis, a type of analysis notorious for its imprecision [52, 37], severely limited the applicability of automatic parallelization.

Following the success of speculation for extracting ILP, speculation in automatic parallelization has gained traction in the last decade [64, 77, 93, 48, 44]. Speculation allows the compiler to optimize for the expected case. The effect is dramatic since there are many fewer dependences in practice than can be proved nonexistent by memory analysis. Memory speculation is a popular speculative enabling transformation that asserts the absence of memory dependences not manifested (or manifested infrequently) during profiling, backing its assertions with runtime checks to initiate misspeculation recovery when necessary. Parallelizing compilers also commonly employ control speculation to simplify the program's control flow by asserting the direction of biased branches and initiating recovery when the speculatively dead path is taken. Another important speculative enabling transformation is speculative privatization proposed in Privateer [44]. With speculation, Privateer is able to handle dynamic data structures even in the presence of unrestricted pointers, a task that proved insurmountable for non-speculative privatization techniques.

Despite the dramatic advance that speculation represents for automatic parallelization, challenges remain that prevent its widespread adoption [16, 32, 73, 46]. While mem-

ory speculation is popular, its relaxed program dependence structure comes with a high cost. Even in cases without any misspeculation, validation of memory speculation requires instrumenting memory operations on every iteration to log or communicate speculative accesses to additional validation code. For large regions with many speculation checks, the validation cost can become prohibitively expensive, negating the benefits of the parallelization. Speculative privatization may also entail high overheads but in a different way. Correctly merging the private memory state of each parallel worker at the end of a loop invocation can require speculative privatization systems to monitor large write sets during execution, significantly degrading their profitability [48, 44, 83]. This dissertation demonstrates that much of these costs result from the lack of speculation-awareness and planning in compiler analysis and optimization.

1.1 Dissertation Contributions

This dissertation shifts the focus from applicability to profitability and proposes a parallelization system that minimizes the overheads of state-of-the-art speculative parallelization approaches while maintaining their applicability. The proposed system, called Per*spec*tive, is an automatic parallelization framework integrating a speculation-aware collaborative analysis framework (SCAF), new efficient variants of speculative privatization, and a new planning phase to select the cheapest set of parallelization-enabling transformations.

1.1.1 Speculation-Aware Collaborative Analysis Framework

The validation and recovery code inserted by speculative transformations can be viewed as dynamically-enforced assertions ensuring that certain data or control flow 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 these 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.

This dissertation aims to enable lower-cost speculation with a modular, collaborative, and speculation-aware memory analysis framework (§5). 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 [43]) 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 that is 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 analysis 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.

Relative to the best prior speculation-aware dependence analysis technique, by maximizing the impact of inexpensive speculation, SCAF dramatically reduces the need for

expensive-to-validate memory speculation in the hot loops of all 16 evaluated C/C++ SPEC benchmarks (§7.1).

1.1.2 Planning & New Enablers

Per*spec*tive's planning phase (§4.2) makes feasible the addition of new efficient speculative enablers (§4.3) by selecting the applied enablers based on their cost and overall impacts. Prior work systems were not sophisticated enough to make these decisions in an informed way. As a consequence, these systems generally had a small number of powerful, but expensive-to-validate, speculative enablers [44, 64].

1.1.3 Fully-Automatic Parallelizing Compiler

This work achieves scalable automatic DOALL parallelization on a commodity shared-memory machine without any programmer hints (§7.2). Per*spec*tive is evaluated on a set of 12 C/C++ benchmarks used in prior state-of-the-art automatic parallelization system papers [44, 48, 15]. On a 28-core machine, Per*spec*tive yields a geomean whole-program speedup of 23.0× over sequential execution. This represents a doubling in performance compared to Privateer, the most applicable prior state-of-the-art speculative-DOALL system [44]. These results come from the effective usage of static properties of the code in conjunction with cheap speculative assertions, the careful selection of applied transformations, and a lightweight process-based runtime system.

1.1.4 Summary

In summary, the primary contributions of this dissertation are:

- A novel **speculation-aware** collaborative dependence analysis framework (SCAF) that computes the full impact of speculation on memory dependence analysis (§5), dramatically reducing the need for expensive-to-validate memory speculation (§7.1);

- A new **planning** phase that combines non-speculative and speculative techniques to select the most profitable set of parallelization-enabling transformations (§4.2);
- New efficient **speculative privatization transformations** that avoid the overheads of prior speculative privatization techniques (§4.3);
- A **fully-automatic** speculative parallelization framework for **commodity hardware** (§6) that exhibits **scalable** speedups by minimizing the speculative parallelization overheads of prior work (§7.2).

These contributions constitute an important and necessary step towards fulfilling the promise of automatic parallelization.

Published work: The speculation-aware collaborative dependence analysis framework (Chapter 5) has been published in [6], while the proposed approach (Chapter 4) and infrastructure (Chapter 6) for automatic parallelization have been published in [5].

Chapter 2

Background

Automatic parallelization has the potential to enable efficient use of multicore systems without undue programmer effort. The programmer writes sequential code, and then a parallelizing compiler automatically produces an executable that runs efficiently in multiple cores. A major challenge for automatic parallelization is handling dependences (§2.1). To address this challenge, state-of-the-art research involves three main components:

(i) Dependence Analysis (§2.2) tries to disprove the existence of dependences;

(ii) Enabling Transformations (§2.3) break dependences, but almost always with a cost;

(iii) New Parallelization Transformations (§2.4) provide ways to tolerate unremovable dependences.

2.1 Dependences

This section provides definitions of various types of dependences (§2.1.1 - §2.1.4) in the context of this dissertation and introduces the program dependence graph, a convenient program representation for parallelizing compilers (§2.1.5).

2.1.1 Memory Dependences

This dissertation adopts the following definition of memory dependence:

Definition 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.

A memory dependence is further classified to:

- *RAW* (read-after-write) dependence (also called "flow" or "true" dependence) when an instruction writes a value that is subsequently read by another instruction.
- *WAR* (write-after-read) dependence (also called "anti" dependence) when an instruction reads a value that is subsequently overwritten by another instruction.
- *WAW* (write-after-write) dependence (also called "output" dependence) when an instruction writes a value that is subsequently overwritten by another instruction.

We say that a dependence is "false" if it is either an anti or an output dependence.

Side-effects: This dissertation models side-effect dependences as memory dependences. For example, calls to print statements have visible effects outside the program's execution context and should not be re-ordered. Memory analysis in this work computes the necessary memory dependences to enforce proper ordering of such calls.

2.1.2 Register Dependences

Representation of the code in Static Single Assignment (SSA) [22] form renders computation of data dependences carried via registers[1] trivial. Registers in SSA only induce true dependences (i.e., no anti or output dependence) since every register has a single definition that dominates all uses.

The compiler infrastructure presented in this dissertation is built upon LLVM. Thus, the compiler's intermediate representation (IR) is in SSA form, allowing trivial computation of dependences carried via the virtual registers of LLVM IR. These register dependences are separated in this dissertation from the rest of data dependences that are carried through memory; memory dependences refer only to the latter.

2.1.3 Control Dependences

This dissertation employs the definition of control dependence from Ferrante et al. [30]:

Definition 2 (Control Dependence)**.** *Let X and Y be two nodes in a control-flow graph. Y is* control dependent *on X iff:*

(i) there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y; and

(ii) X is not strictly post-dominated by Y.

This definition is restricted to control-flow graphs where the "end" node is reachable by all nodes. To correctly compute control dependences for programs with function calls that may not return, this work inserts auxiliary control-flow edges from such function calls to the program's "end" node, as in [72].

[1] In this dissertation, registers refer to storage locations that are accessed only directly through a unique name and cannot be accessed indirectly through a pointer.

2.1.4 Intra- & Cross-Iteration Dependences

Static instructions often represent several dynamic instances during program execution. For parallelization techniques, it is essential to disambiguate between certain dynamic instances of instructions within loops. In particular, prior work in parallelization [71, 78, 45, 42] considers two types of dependences: *cross-iteration* (Definition 3) and *intra-iteration* (Definition 4). This separation is necessary since most parallelization techniques can tolerate intra-iteration dependences while cross-iteration depedences either prohibit parallelization or increase communication costs (§2.4). In other words, without this separation, more dependences than necessary would restrict parallelization.

Definition 3 (Cross-iteration Dependence)**.** *There is a* cross-iteration *dependence with respect to loop L from instruction t to instruction u iff*

(i) there is a dynamic instance t_i of t which executes during the i-th iteration of L and

(ii) a dynamic instance u_j of u which executes during the j-th iteration of L,

(iii) such that $i \neq j$ and there is a dependence from t_i to u_j.

In this context, a dynamic instance of a static instruction represents all the executions of this instruction in a given loop iteration.

Definition 4 (Intra-iteration Dependence)**.** *There is an* intra-iteration *dependence with respect to loop L from instruction t to instruction u iff*

(i) there are dynamic instances t_i, u_i of t, u, respectively, which both execute during the i-th iteration of L,

(ii) such that there is a dependence from t_i to u_i.

2.1.5 Program Dependence Graph

A program dependence graph (PDG) [30] is a graph with static instructions as vertices and control, register, and memory dependences among these instructions as directed edges. In

this dissertation, each pair of instructions might be connected with more than one dependence edge. Intra-iteration and cross-iteration dependence edges (§2.1.4) are separated, and different edges are created for each type of memory dependence (flow, anti, and output dependence).

A PDG is a convenient program representation for code transformations, and especially for thread-level parallelism. To be correct, transformations only need to respect all dependences depicted in the PDG since transforming an input PDG while respecting its dependences produces an isomorphic PDG, and Horwitz et al. [38] prove that two programs are strongly equivalent if they have isomorphic PDGs.

2.2 Dependence Analysis

Dependence 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.

This dissertation focuses on handling and mitigating memory dependences, which pose one of the biggest challenges for automatic parallelization systems. Therefore, in the context of this dissertation, dependence analysis will refer to the static analysis of memory dependences, and the terms dependence analysis, memory analysis, and memory dependence analysis will be used interchangeably.

The goal of dependence analysis algorithms is to disprove dependences among program operations. To that end, algorithms attempt to invalidate one of the conditions for the existence of a memory dependence (Definition 1): disproving *aliasing*, disproving a *feasible path*, or proving a *killing operation* exists along all feasible paths. If they fail to disprove at least one condition, they conservatively report that those operations *may* depend.

Decades of research have been devoted to increasing the precision of memory analysis to address each of these conditions. Advancements include algorithms in points-to analysis [4, 9, 12, 55, 57, 88], alias analysis [99, 60], shape analysis [86, 34, 35], and loop dependence analysis [8, 76]. Nevertheless, memory analysis is undecidable [52] and remains insufficiently precise in practice, especially for languages like C/C++ [37].

2.3 Enabling Transformations

Parallelizing compilers apply enabling transformations to make the program more amenable to parallelization.

2.3.1 Speculation

Speculation allows optimizations to overcome the limitations of static analysis. Speculation typically relies on profiling information to identify data and control flow relationships expected to rarely or never occur during program execution.[2] Offline runs of the target program with representative inputs produce this profiling information. Relationships reported by static 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.

Speculative optimizations optimize for the common case by assuming that these speculative assertions are true while transforming the code. To preserve correctness, speculative optimizations add checks to activate recovery code when these assumptions prove untrue. Recovery involves rolling back to a previous valid program state and non-speculative re-execution. 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 check-

[2] This dissertation refers to predictions based on profiling information as *speculative assertions*

ing for misspeculation, a cost that exists even when there is no misspeculation. More concretely, the use of a speculative optimization is profitable when:

$$\begin{aligned} \textit{original\_time} > (1 - \textit{misspec\_rate}) \times (\textit{validation\_time} \ + \textit{optimized\_time}) + \\ (\textit{misspec\_rate}) \times (\textit{invalid\_time} + \textit{recovery\_time}) \end{aligned}$$

The *optimized_time* denotes the execution time of the optimized code without considering any speculation overhead. The *original_time* denotes the execution time of the code without the application of the speculative optimization. The *invalid_time* denotes the execution time till the detection of misspeculation and involves the work that gets undone with the rollback. Checkpointing (periodically saving a valid program state) during speculative execution increases the *validation_time* but decreases the *invalid_time* and the *recovery_time*.

Aggressive use of speculation is most prominently observed in parallelization schemes, where high-performance gains can compensate for the overheads introduced by speculation [82, 44, 48, 94, 64].

Memory Speculation: Memory speculation is the most commonly used and powerful speculation technique [82, 94, 64, 44, 48]. It asserts the absence of memory dependences non-observed during profiling using a loop-sensitive memory dependence profiler [18]. Yet, memory speculation is also the most expensive speculation technique. Excessive usage of memory speculation often negates its enabling effect [32, 16, 93]. 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 [44, 69, 82]. This is expensive for software-only systems where monitoring of large read and write sets results in dramatic slowdowns [44, 77].

```
 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 2.1: Speculation validation code examples. Validation of inexpensive speculative assertions involves only a few bitwise/arithmetic/branch instructions, while the memory speculation check involves many more operations, including memory accesses.

To lower the validation cost, a diverse set of cheaper but less powerful speculation techniques have been proposed. Prior work includes speculative assertions that: certain control paths are never taken; certain load instructions always read the same value; certain memory objects live only within a single loop iteration [44, 48, 93]; pointers only reference a restricted family of objects [44]; certain types of data structures change infrequently [79]; or that certain memory objects are read-only [44].

Figure 2.1b shows an assembly code snippet of a typical memory speculation validation code and compares it with an example of a cheap-to-validate speculation (points-to heap/family check in Figure 2.1a). Validation of the rest of the aforementioned cheap-to-validate speculative assertions is not more complicated than the simple check shown in Figure 2.1a. Note that if a parallelization transformation employs memory speculation for a cross-iteration dependence, memory speculation validation additionally involves the communication of memory footprints among parallel workers. In contrast, for most inexpensive speculation techniques, each parallel worker only needs to perform local checks with no communication overhead.

2.3.2 Privatization & Reduction

The reuse of data structures introduces artificial constraints to automatic parallelization. Privatization is an enabling transformation that creates private data copies for each parallel worker to remove contention caused by the reuse of data structures. The privatization criterion in the context of loop parallelization follows:

Definition 5 (Privatization Criterion [44])**.** *We say a memory object M is privatizable in a loop L if M does not partake in cross-iteration flow dependences, namely there is no read of M in iteration i of L that should return a value written in an earlier iteration j.*

Reduction is another important parallelization-enabling transformation. Reduction expands storage locations to relax ordering constraints on associative and commutative operations. Contrary to privatization, there is a real flow dependence in reducible operations, but it is bypassed by expanding the shared storage location in multiple copies. Each parallel worker updates independently its own private copy during the execution of the loop. At the end of the loop invocation, all copies are merged together to form the final result. This dissertation's applicability criterion of reduction follows:

Definition 6 (Reduction Criterion [44])**.** *We say a memory object M is reducible in a loop L if all writes to M within L are performed with a single associative and commutative operator, and no other operation within L reads an intermediate value of M.*

Speculative Variants Early work focused on satisfying the privatization and reduction criteria statically [96, 28]. To increase the applicability of these enablers, later work implemented speculative variants [82, 23]. However, these works are limited to arrays and scalar variables and cannot handle dynamically allocated data structures and pointers; hence they are largely inapplicable to most C/C++ programs. More recently, Privateer [44] introduced more advanced and applicable speculative variants of privatization and reduction that are insensitive to memory layout by employing profile-guided speculative separation of mem-

ory objects. These variants are powerful enough to handle dynamic and recursive data structures, even in the presence of unrestricted pointers.

2.4 Parallelization Transformations

After the application of enablers, the program is parallelized, if possible. Many parallelization transformations with varying degrees of applicability and effectiveness have been proposed in the literature. The most desirable and simple is DOALL [47]. Other techniques, like DOACROSS [21] or PS-DSWP [78], are less efficient but more applicable thanks to their ability to tolerate cross-iteration dependences. This dissertation focuses mainly on DOALL parallelization. However, some proposed techniques are agnostic to the parallelization scheme (§5), and the proposed infrastructure, apart from DOALL, also supports pipelined parallelization [78, 71] (§6.2). Thus, this background section also provides a brief discussion of other parallelization techniques.

2.4.1 DOALL Parallelization

DOALL parallelization is applicable to a loop when each iteration of this loop is independent of the other iterations. In other words, it is applicable for loops that do not have any cross-iteration dependences. If this criterion is satisfied, the DOALL transformation can schedule each loop iteration in parallel. Since there is no need for communication among the parallel workers, DOALL is the most profitable parallelization technique. The critical path of the loop is just the execution latency of the longest single iteration. Since even one cross-iteration dependence renders DOALL inapplicable, multiple research works [82, 44, 48, 64], including this dissertation, try to eliminate every single cross-iteration dependence by any means possible (e.g., sophisticated static analysis, new enabling transformations).

2.4.2 Tolerating Cross-Iteration Dependences

Loops in general-purpose programs sometimes have unremovable cross-iteration dependences that prevent the use of DOALL. Figure 2.2a shows a commonly found code pattern with a cross-iteration dependence. This code pattern involves traversing through a linked list and performing some work on every node of the linked list. Note that the critical path (i.e., longest dependence chain) for this loop comprises of all A_i and B_i instructions, where i represents the i_{th} loop iteration.

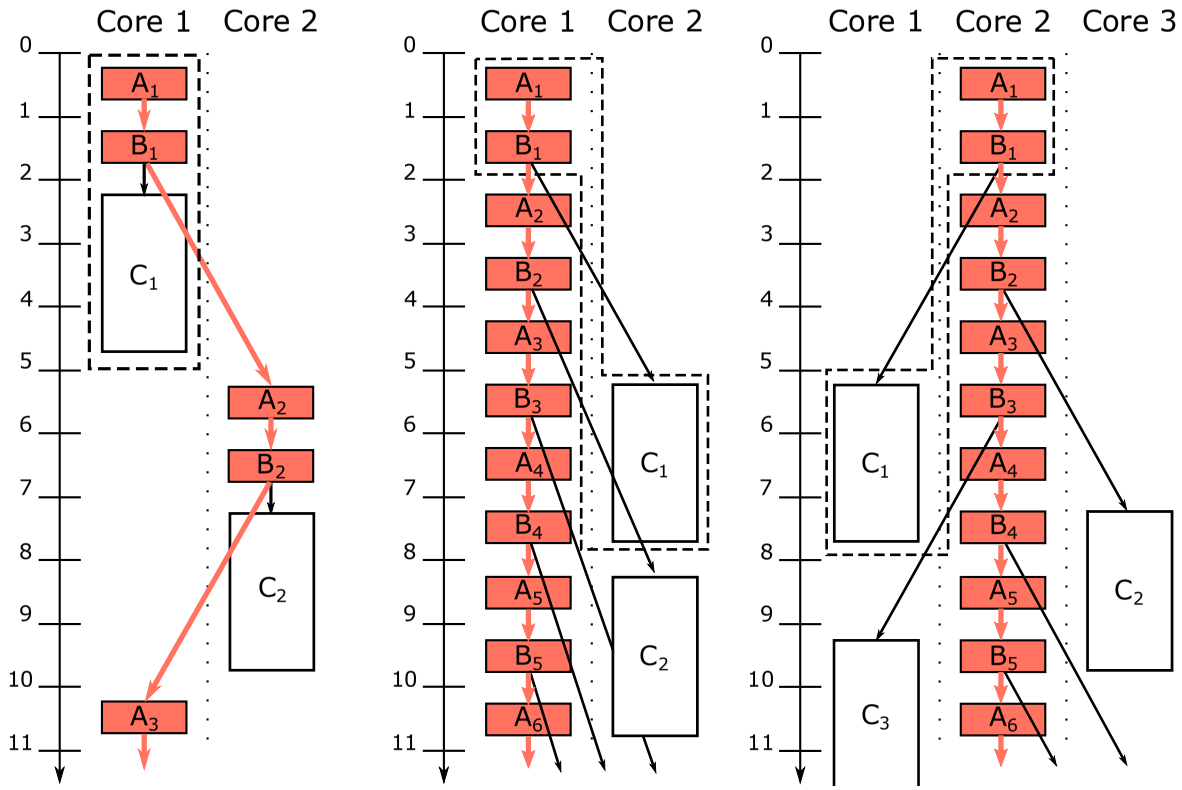
Historically, one way to parallelize such loops is to use DOACROSS, where each loop iteration is scheduled on a separate core, and cross-iteration dependences are forwarded between cores [47]. Figure 2.2b shows the DOACROSS parallelization for this loop, which takes an average of 5 cycles per loop iteration to execute. As can be seen from the figure, the cross-iteration dependence that is a part of the critical path ($A_i \rightarrow B_i \rightarrow A_{i+1}$) must be communicated across cores. This results in a cyclic communication pattern that increases the length of the critical path by putting the latency of the inter-core communication onto the critical path. The critical path now consists of the latency of execution of all A_i and B_i instructions and the inter-core communication latency (set to 3 cycles in this example) required to communicate each $B_i \rightarrow A_{i+1}$ dependence. If the inter-core communication latency were to be increased from 3 cycles to 4 cycles, execution of the loop would slow down from 5 to 6 cycles per iteration.

Campanoni et al. [15] proposed HELIX to perform a finer-grained and more scalable parallelization than its DOACROSS predecessor. HELIX carefully overlaps inter-core communication with computation in an attempt to mitigate DOACROSS's sensitivity to inter-core communication latency. HELIX is especially successful at moving data forwarding off the critical path when proper hardware support is provided (HELIX-RC [13]) or minor output distortions are acceptable (HELIX-UP [14]).

To maximize loop performance, one needs to minimize the latency of the critical path of the loop. A minimal path can quickly free up parallel work in the noncritical sections of

```
A: while(node->next){
B:   node=node->next;
C:   work(node);
   }
```

(a) Linked list traversal code example



(b) DOACROSS [21] 5 cycles/iteration

(c) DSWP [80, 71] 3 cycles/iteration

(d) PS-DSWP [78] 2 cycles/iteration

Figure 2.2: Comparison of parallelization transformations that can tolerate cross-iteration dependences. Edges connecting instructions from different iterations represent cross-iteration dependences. Colored boxes and thick edges highlight the critical path. Inter-core communication latency is set to 3 cycles. A similar figure appears in [31].

the loop, thereby yielding better performance. Based on this insight, Decoupled Software Pipelining (DSWP) was proposed [80, 71]. Instead of scheduling the loop iteration as a single monolithic unit, DSWP divides the loop body into multiple pipeline stages and assigns each stage to a different core for execution. DSWP schedules the critical path of the loop on the same core. Some values corresponding to dependences between instructions must still be communicated between cores, but none of these dependences are on the loop's critical path. Figure 2.2c shows how DSWP parallelizes the loop. Unlike DOACROSS or HELIX, the scheduling of dependences on the critical path as core-local gives DSWP communication-latency tolerance. Thus, even if the latency of the inter-core communication were to increase, DSWP would still take on average 3 cycles per loop iteration. However, despite the latency tolerance, DSWP's scalability is restrained to the number of pipeline stages.

In response, Raman et al. [78] developed Parallel Stage Decoupled Software Pipelining (PS-DSWP), an enhancement to DSWP. PS-DSWP showed that the noncritical instructions scheduled by DSWP on different cores may subsequently be parallelized in a myriad of ways, including DOALL [78] and LOCALWRITE [40]. Figure 2.2d shows that the PS-DSWP transformation executes each loop iteration of this example in only 2 cycles once the pipeline is filled. This transformation detects that there are no cross-iteration dependences between the noncritical path instructions and thus schedules these instructions on different cores. Therefore, by isolating the critical path of a loop in the first stage and subsequently parallelizing the noncritical work sections, PS-DSWP unlocks scalable parallelism for loops previously thought to be inherently sequential [49, 78].

Chapter 3

Motivation

Since this dissertation focuses mainly on DOALL parallelism, this chapter first discusses the state-of-the-art for DOALL parallelization systems. It then presents the inefficiencies of the state-of-the-art to motivate this dissertation's work. Note that the presented inefficiencies are not unique to DOALL and actually plague any type of prior automatic parallelization system. Similarly, the presented techniques that address these inefficiencies are either agnostic to the parallelization scheme (§5) or are extensible to schemes beyond DOALL (§6.2).

3.1 State-of-the-Art for DOALL Parallelization

Software-based automatic DOALL parallelization systems have been studied for a long time. Early works in the 1990s, including Polaris [10], SUIF [100], PD [81], and Hybrid Analysis [85], use static or runtime analysis to parallelize programs and examine the applicability of enabling transformations such as privatization and reduction. However, the imprecision of static analysis and the difficulty of extracting low-cost runtime checks limit the applicability of these systems to scientific codes.

More recent works like STMlite [64], Cluster Spec-DOALL [48], Privateer [44] use profile-guided speculation to overcome the limitations of static analysis and enable paral-

lelization of loops with pointers, irregular memory accesses, and complex control flows. Among these works, Privateer [44] supports speculative privatization and reduction even in the presence of unrestricted pointers by using speculative heap separation, and it has the greatest applicability of all prior automatic speculative-DOALL systems.

Despite increased applicability, evaluation results of automatic speculative-DOALL systems on real hardware are still underwhelming due to overheads that often negate the benefits of parallelization [16, 32]. This dissertation first identifies core inefficiencies of the state-of-the-art parallelizing compilers and subsequently describes how the proposed frameworks mitigate them.

3.2 Overheads of State-of-the-Art

Privateer [44] is the most applicable automatic DOALL parallelization system, and thus this dissertation focuses on the parallelization overheads of Privateer. By examining the evaluation of Privateer, this work identifies two core inefficiencies:

(i) excessive use of memory speculation, which is the most common problem of prior speculative parallelization systems; and

(ii) expensive privatization, which applies to most systems with privatization support, especially speculative ones.

In §3.2.1 and §3.2.2, this dissertation discusses the impact of these two main overheads on Privateer along with a motivating example taken from the `dijkstra` benchmark (used in Privateer's evaluation) from MiBench [36]. The simplified code for the hot loop of this benchmark is shown in Figure 3.1. Section 4.4 shows how the proposed approach in this dissertation avoids these overheads for this example.

```
1  int *pathcost; // dyn alloc 1-D N
2  int *adj; // dyn alloc 2-D NxN
3  int dist;
4  int nDist;
5
6  void allocatePathCost() {
7    pathcost = (int*)malloc(N*sizeof(int));
8  }
9
10 int dequeue() {
11   if (!nullQHead()) {
14     dist = ...
15     ...
16   }
19 }
20
21 void hot_loop(int N) {
26   for (src=0; src<N; src++) {
29     for (i=0; i<N; i++)
30       pathcost[i] = inf;
31
32     enqueue(src, 0);
33     while (!emptyQ()) {
34       int v = dequeue();
35       for (i=0; i<N; i++) {
39         nDist = adj[v][i] + dist;
42         if (pathcost[i] > nDist) {
45           pathcost[i] = nDist;
46           enqueue(i, nDist);
47         }
48       }
49     }
53   }
55 }
```

Figure 3.1: Motivating example from `dijkstra` [36]

3.2.1 Excessive Use of Memory Speculation

Privateer's excessive use of memory speculation leads to large overheads for monitoring speculative memory accesses, with an average of 23.7 GB of reads and 18.4 GB of writes monitored per benchmark, as reported in the paper [44]. This problem is especially apparent for the `dijkstra` benchmark which has particularly high overheads (84.9GB of reads and 56.7GB of writes) that limit its speedup to 4.8× on 24 cores. For example, Privateer resorts to memory speculation to resolve the cross-iteration flow dependence from line **14** to line **39** in order to privatize **`dist`**. Static analysis alone is unable to disprove this dependence, since the write to **`dist`** is inside a conditional block. Other prior speculative parallelization systems, similarly to Privateer, cannot avoid the use of memory speculation in this case.

As this dissertation demonstrates (§7), prior work resorts to memory speculation due to: (i) lack of awareness within the compiler of the full effect of cheaper-to-validate speculative techniques; and/or (ii) lack of planning that would avoid unnecessary use of expensive speculation and prioritize cheaper alternatives. By enabling judicious use of speculation with the introduction of speculation-aware analysis and a planning phase, this work manages to remove the aforementioned dependence in this example without the use of expensive-to-validate memory speculation, as shown in §4.4.

3.2.2 Expensive Privatization

Perhaps unexpectedly, parallelized programs may still have large overheads due to bookkeeping of writes to privatized objects, even without the use of memory speculation. For `dijkstra`, even assuming checks for speculative reads are removed, Privateer still needs to log 56.7 GB of writes to privatized objects, which constitutes around 20% of each parallel worker's time. In Figure 3.1, static analysis alone can disprove all cross-iteration flow dependences related to **`pathcost`** and safely privatize it. However, because **`pathcost`** is a live-out object (i.e., might be read after the loop invocation), Privateer still logs its writes to track in which iteration each byte of the object was last written.

Prior work, including Privateer, lacks the decision-making mechanisms to simultaneously consider multiple applicable enabling transformations, resorting to a few powerful, but expensive speculative enablers. This work introduces new efficient speculative privatization variants that are enabled by its planning phase. By selecting a more efficient privatization variant, this work avoids the unnecessary bookkeeping for the **`pathcost`** object in this example, as discussed in §4.4.

Chapter 4

The Per*spec*tive Approach

For years, the biggest problem of automatic parallelization was limited applicability. Progress on that front was made, mainly with the introduction of speculation, rendering automatic parallelization more applicable. Given that success, this dissertation shifts the focus to profitability by making those approaches to applicability more profitable.

Current parallelizing compiler designs utilize memory analysis and speculative techniques independently and apply a fixed sequence of powerful but often high-overhead enabling transformations. These designs lead to overheads that often negate the benefits of parallelization. As one of its contributions, this dissertation identifies many such overheads as being unnecessary.

To overcome the inefficiencies of prior work, this dissertation introduces a parallelization framework, called Per*spec*tive, that integrates a novel speculation-aware dependence analysis framework (§4.1) and a planning phase involving careful selection of applied transformations (§4.2). Per*spec*tive's design enables the discovery of efficient parallelization opportunities not possible in prior parallelizing compilers, including new efficient speculative privatization transformations (§4.3).

4.1 Speculation-Aware Analysis

Prior techniques use memory analysis and speculative assertions independently. This is problematic since the unrecognized effect of speculative assertions can be significant. Instead, this work proposes a speculation-aware collaborative analysis framework (SCAF) that combines the strengths of static analysis and cheap-to-validate speculative assertions to reduce the need for expensive speculation. SCAF enables memory analysis to view cheap-to-validate speculative assertions as facts, ignoring the possibility of misspeculation. To enable the compiler to preserve correctness and judiciously speculate, SCAF reports for each analysis response the speculation assertions leveraged in the process. Thus, the compiler only needs to protect (with validation and recovery code) useful speculative assertions while fully leveraging each one of them. For the design and implementation details of the speculation-aware analysis framework refer to §5.

4.2 Planning

Unlike prior speculative systems that apply a fixed sequence of parallelization-enabling transformations, Per*spec*tive proposes a more *sensible* approach: before applying any transformation, plan first.

Enabling transformations modify the code to remove parallelization inhibitors, which in the context of DOALL parallelization are cross-iteration dependences. All transformations are split into two parts to facilitate planning: the applicability guard that participates in the planning phase of the compilation, and the actual transformation that is applied, if selected, in the transformation phase. The applicability guard utilizes properties produced by the speculation-aware analysis framework to determine which parallelization inhibitors the corresponding transformation can handle. The interface between the speculation-aware analysis framework and enabling transformations is an annotated program dependence graph (PDG). The output of each applicability guard is collected in a transformation proposal.

Each proposal also includes the cost for the application of the transformation and a set of speculative assertions required for the transformation to be applicable. At the end of the planning phase, the *transformation selector* picks a minimal-cost set of transformation proposals necessary for DOALL parallelization. This planning approach generalizes beyond DOALL, as discussed in §6.2.

Instead of targeting individual cross-iteration dependences, memory-related transformations offer to handle a set of memory objects, effectively addressing all associated cross-iteration dependences. This object-centric approach is motivated by the fact that memory-related enabling transformations often operate at the object level. For example, the privatization transformation creates private copies of memory objects.

4.3 New Enabling Transformations

Prior work on speculative parallelization focuses on the enabling effects of transformations without enough consideration of their costs. To maximize applicability, enabling transformations are often given a program dependence graph relaxed with the use of all the available speculative assertions. This approach not only creates ambiguity in terms of which speculative assertions are necessary but also prevents the usage of efficient variants of transformations. By exposing a combination of static analysis information and the effect of speculative assertions, Per*spec*tive enables more efficient enabling transformations. This is especially true for the case of speculative privatization, efficient variants of which are explored in this dissertation.

Prior software speculative systems with extended support for privatization [44, 48] only infer the basic privatization property: a memory object does not have any cross-iteration data flows. This way, speculative privatization application involves costly instrumentation of all write accesses of privatized objects for logging or communication. At commit, the

private copies of each worker are merged according to metadata that specifies which worker last wrote each byte.

To avoid expensive monitoring of write sets during parallel execution and minimize copy-out costs, this dissertation proposes four efficient variants of the speculative privatization transformation. These variants require additional memory object properties apart from the basic privatization property to be applicable. With respect to a target loop, a private memory object can additionally be:

(i) *independent*, if it does not source or sink any cross-iteration false dependences;

(ii) *overwritten*, if it is written to the same memory locations at every loop iteration;

(iii) *predictable*, if the live-out content of the private object is predictable; or,

(iv) *local*, if it is allocated outside the loop but all its accesses are contained within the loop.

Inference of any of these four private properties allows the complete elimination of bookkeeping costs provided that the basic privatization property was satisfied without the use of memory flow speculation. The first two variants have been explored by Tu et al. [96] but were limited to static-analysis-based detection of privatization. Unlike any prior work, Per*spec*tive extends the applicability of these two variants to programs with pointers, dynamic allocation, and type casts using speculative assertions.

Figure 4.1 demonstrates cases where the proposed privatization variants are applicable. All the memory objects in this example satisfy the privatization criterion. However, all objects except for the **privatized_array** satisfy one additional property that enables efficient privatization. Prior work cannot distinguish these different cases and thus would have to apply speculative privatization with expensive monitoring for all the arrays in this example.

```
1  int *independent_array; // dyn alloc 1xN
2  int overwritten_array[K];
3  int predictable_liveout_array[K] = {0};
4  int local_array[K];
5  int privatized_array[K];
6
7  // parallelized loop
8  for (i=0; i<N; i++) {
9
10   independent_array[i] = ...
11
12   for (j=0; j<K; j++) {
13
14     if (!rare)
15       overwritten_array[j] = ...;
16
17     if (rare)
18       predictable_liveout_array[j] = ...;
19
20     if (non_biased_cond) {
21       local_array[j] = ...;
22       privatized_array[j] = ...;
23     }
24     ...
25   }
26 }
27
28 // no use for local_array
29 // outside the parallelized loop
30 ...
```

Figure 4.1: Example where all proposed speculative privatization variants are applicable. i) *independent*: the **independent_array** can be shared among the workers (executing outer-loop iterations) since there are no overlapping memory accesses. ii) *overwritten*: the **overwritten_array** is speculatively fully-overwritten at every outer-loop iteration (speculated that the rare condition is never true); thus its liveout state is the last iteration's state of the array. iii) *predictable*: the **predictable_liveout_array** is predicted to include only zeroes at the end of the loop invocation (speculated that the rare condition is never true). iv) *local*: no use for the **local_array** outside the parallelized loop; thus no need to compute its liveout state. v) *just privatizable*: the **privatized_array** has unknown liveout state; thus its writes need to be monitored.

4.4 Example

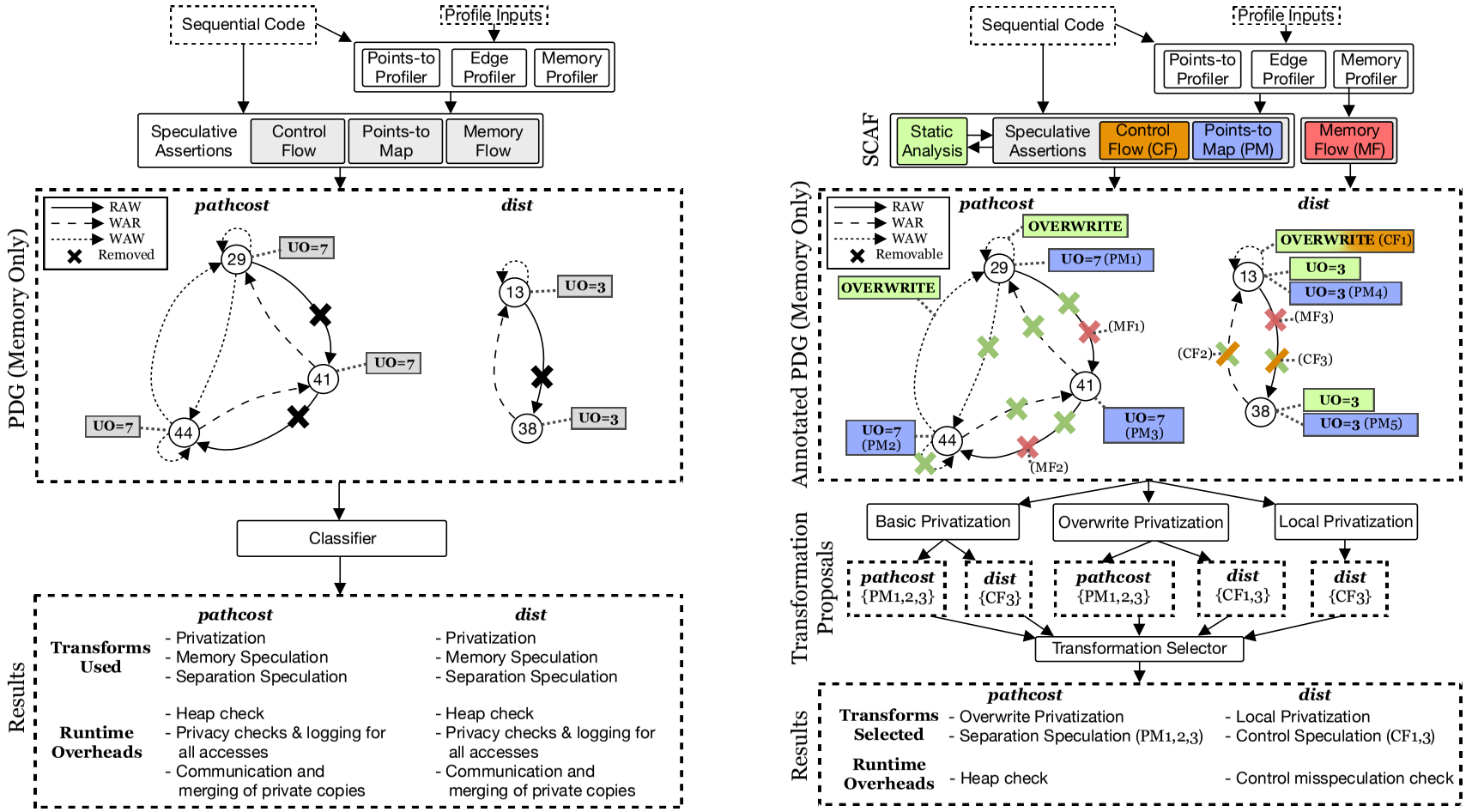
This section describes how the new ideas introduced in this dissertation (§4.1, §4.2, §4.3) enable efficient parallelization of the `dijkstra` benchmark from MiBench [36]. This section also highlights the limitations of prior work. Consider again the motivating example discussed in §3.2 and presented again in Figure 4.2. Figure 4.3 compares the compilation flow of Per*spec*tive with that of Privateer [44] for handling memory objects **`pathcost`** and **`dist`** from this code example.

```
1  int *pathcost; // dyn alloc 1-D N
2  int *adj; // dyn alloc 2-D NxN
3  int dist;
4  int nDist;
5
6  void allocatePathCost() {
7    pathcost = (int*)malloc(N*sizeof(int));
8  }
9
10 int dequeue() {
11   if (!nullQHead()) {
14     dist = ...
15     ...
16   }
19 }
20
21 void hot_loop(int N) {
26   for (src=0; src<N; src++) {
29     for (i=0; i<N; i++)
30       pathcost[i] = inf;
31
32     enqueue(src, 0);
33     while (!emptyQ()) {
34       int v = dequeue();
35       for (i=0; i<N; i++) {
39         nDist = adj[v][i] + dist;
42         if (pathcost[i] > nDist) {
45           pathcost[i] = nDist;
46           enqueue(i, nDist);
47         }
48       }
49     }
53   }
55 }
```

Figure 4.2: Sequential version of the motivating example from `dijkstra` [36]. The line numbering is consistent with the parallelized versions of this example presented in Figure 4.4 (missing line numbers account for inserted logging and checking operations in the parallelized code).

Per*spec*tive employs an exploration phase that yields a much more profitable plan than Privateer's scheme. At first, the **speculation-aware collaborative analysis framework (SCAF)** processes the sequential code and the profile information, and it is queried to produce a program dependence graph (PDG) annotated with properties and underlying speculative assertions, as shown in Figure 4.3b. Observe that memory flow speculation is not included in SCAF, since memory speculation just asserts the absence of dependences without being able to collaborate with other speculation or memory analysis modules within SCAF.

In this example (Figure 4.2), the branch in line **11** is heavily biased; it is always taken in practice. Therefore, control speculation leveraging edge profiling information can assert



(a) Privateer [44]

(b) Per*spec*tive (This Work)

Figure 4.3: Comparison of the decision process for handling memory objects **`pathcost`** and **`dist`** of `dijkstra`. Numbers in circles are line numbers in Figure 4.2; "UO=#" means the underlying object is allocated/declared in line #; Colors in (b) indicate the component(s) that inferred a property; Privateer does not keep track of how a property was inferred.

that the branch is always taken and that the instruction in line **14** executes at every invocation of the *dequeue* function. Normally, memory analysis is unaware of this assertion and thus is unable to handle the cross-iteration data flow from line **14** to line **39**. However, the proposed speculation-aware analysis framework allows memory analysis algorithms to view this assertion as a fact, ignore the branch and the possibility of misspeculation, and observe the read in line **39** as being dominated by the write in line **14**. This way, a kill-flow analysis algorithm aware of this assertion can infer that there is no cross-iteration data flow between these two operations since the write to **dist** appears to always kill the flow from a previous iteration before it reaches the read operation in line **39**. This combination of control speculation and static analysis removes a dependence that would require the use of expensive-to-validate memory speculation in any prior speculative parallelization system.

The speculation-aware analysis framework is not only queried to produce different options for removable edges but also to provide useful information for non-removable edges. Non-removable output dependence edges, in the example in Figure 4.3b, are annotated with the *overwrite* property that indicates that the destination operation always overwrites the footprint of the source operation.

Furthermore, the speculation-aware analysis framework is queried to annotate instructions that may access memory with underlying memory objects, namely the instruction's memory footprint ("UOs" in Figure 4.3b). This information can be inferred either statically or via points-to profile information, and it is required by privatization transformations that need to map memory objects to memory operations for correct identification of privatizable objects.

In the next step, based on the annotated PDG, three different transformations offer to handle memory objects, including **overwrite privatization** and **local privatization**, introduced in §4.3.

To enable DOALL parallelization, the **transformation selector** then selects the lowest cost option for each memory object. For example, the *local* privatization's offer is selected

for **`dist`** since it is the cheapest privatization transformation (no monitoring and no copy-out costs), and its needed speculative assertion is the same as those of other options. The **`nDist`** object (not shown in this figure) is also handled by *local* privatization, while the **`pathcost`** array is handled by the *overwrite* privatization transformation.

On the other hand, Privateer, the prior automatic DOALL parallelization system with the highest applicability, cannot parallelize *dijkstra* as efficiently as Per*spec*tive (Figure 4.3a). The problem is that Privateer relies on profile information to create a speculatively relaxed PDG, creating ambiguities on how each dependence was removed. Moreover, the produced PDG does not contain any information on the remaining edges. This overall lack of information impedes consideration of the efficient privatization variants described in this dissertation. Instead, Privateer's approach necessitates excessive memory speculation validation to conservatively preserve program correctness and expensive privatization of the objects with privacy checks and costly merging of private copies. The end result for Privateer's parallelization is high runtime overheads.

Figure 4.4 compares the resulting parallelized versions (in a simplified form) by Per*spec*tive and Privateer. The code includes all the added checks, logging, and live-out handling overheads. The code changes are marked with the average added overhead compared to each worker's useful work. Based on this figure, it is clear that Per*spec*tive is able to parallelize `dijkstra` without Privateer's overheads, thanks to the use of the speculation-aware analysis framework, careful selection of applied transformations, and use of efficient privatization variants. In fact, Per*spec*tive exhibits 4.8× speedup over Privateer for the `dijkstra` benchmark (see §7.2.2).

```
int dequeue() {
  if (!nullQHead()) {
    // Privacy Local Check & Logging
    private_write(&dist, sizeof(int), md); // < 1% added OH
    dist = ...
    ...
  }


}

void worker_loop(int start, int N, int step) {
  void *md = alloc(); // allocate metadata
  // Separation Local Check
  check_heap(pathcost, PRIVATE); // < 0.001% added OH
  check_heap(&dist, PRIVATE); // < 0.001% added OH
  for (src=start; src<N; src+=step) {
    // Privacy Local Check & Logging
    private_write(pathcost, N*sizeof(int), md); // < 1% added OH
    for (i=0; i<N; i++)
      pathcost[i] = inf;

    enqueue(src, 0);
    while (!emptyQ()) {
      int v = dequeue();
      for (i=0; i<N; i++) {
        // Privacy Local Checks & Logging
        private_read(&dist, sizeof(int), md); // 11.4% added OH
        private_write(&nDist, sizeof(int), md); // 13.3% added OH
        nDist = adj[v][i] + dist;
        // Privacy Local Check & Logging
        private_read(&pathcost[i], sizeof(int), md); // 11.6% added OH
        if (pathcost[i] > nDist) {
          // Privacy Local Check & Logging
          private_write(&pathcost[i], sizeof(int), md); // <1% added OH
          pathcost[i] = nDist;
          enqueue(i, nDist);
        }
      }
    }
    if (checkpointDue()) {
      checkPrivAcccessesConflicts(md); // < 1% added OH
    }
  }
  communicateLiveOutMemState(md); // < 1% added OH
}
```

(a) Privateer [44]

```
int dequeue() {
  if (!nullQHead()) {


    dist = ...
    ...
  }
  else
    misspec("Control misspec in dequeue()"); // 0% added OH
}

void worker_loop(int start, int N, int step) {

  // Separation Local Check
  check_heap(pathcost, OVERWRITE_PRIVATE); // < 0.001% added OH

  for (src=start; src<N; src+=step) {


    for (i=0; i<N; i++)
      pathcost[i] = inf;

    enqueue(src, 0);
    while (!emptyQ()) {
      int v = dequeue();
      for (i=0; i<N; i++) {



        nDist = adj[v][i] + dist;


        if (pathcost[i] > nDist) {


          pathcost[i] = nDist;
          enqueue(i, nDist);
        }
      }
    }
  }
  // only last iteration's pathcost array
  // needs to be communicated
  if (isLastIter(src, start, N, step))
    communicate_pathcost(); // < 1% added OH
}
```

(b) Per*spec*tive (This Work)

Figure 4.4: Comparison of parallelized code for `dijkstra`. Logging and checks during loop execution dominate the overheads, indicated in the code as "added OH".

Chapter 5

Speculation-Aware Collaborative Analysis Framework

One of the primary contributions of this dissertation is SCAF, a dependence analysis framework that enables collaboration between memory analysis and speculative techniques without sacrificing modularity. This chapter first motivates the need for a collaborative, modular, and speculation-aware dependence analysis framework and then describes the design and implementation of SCAF.

5.1 Motivation

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 dissertation.

One might attempt to express the effect of speculative assertions by transforming the code. For example, Neelakantam et al. [66] propose converting biased branches to assertions to expose speculative control flow information to subsequent transformations. This approach does not generalize for all the types of speculative assertions, especially for those that assert the absence of certain data flow relationships. For example, the impact of sep-

aration speculation [44] and memory speculation (§2.3.1) 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 transformation may limit the applicability of some subsequent transformations (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 [98, 103, 64, 48]. This dissertation characterizes 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 (§5.1.1).

Monolithic Integration: In this approach, memory analysis algorithms are extended with knowledge and interpretation of profile-based speculative information [29, 26]. 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 in terms of engineering effort and hinders extensibility and maintainability.

Table 5.1: 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 [29, 26]	✗	✓	✗
Composition by Confluence [98, 103, 64, 48]	✗	✗	✓
Composition by Collaboration (**This Work**)	✓	✓	✓

Composition by Collaboration: Motivated by the deficiencies of prior work, this dissertation 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 5.1 summarizes the comparison of this work with existing proposals of the designs described earlier in this section.

5.1.1 Example

For the code example in Figure 5.1, a client wants to determine whether there is a cross-iteration dependence from instruction $i3$ to $i2$. By inspecting the code, one can observe a cross-iteration data flow from $i3$ to $i2$ when the branch is taken. However, since this path

```
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 5.1: Motivating Code Example

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 §2.1.1 (*alias*, *update*, *feasible-path*, *no-kill*) 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.

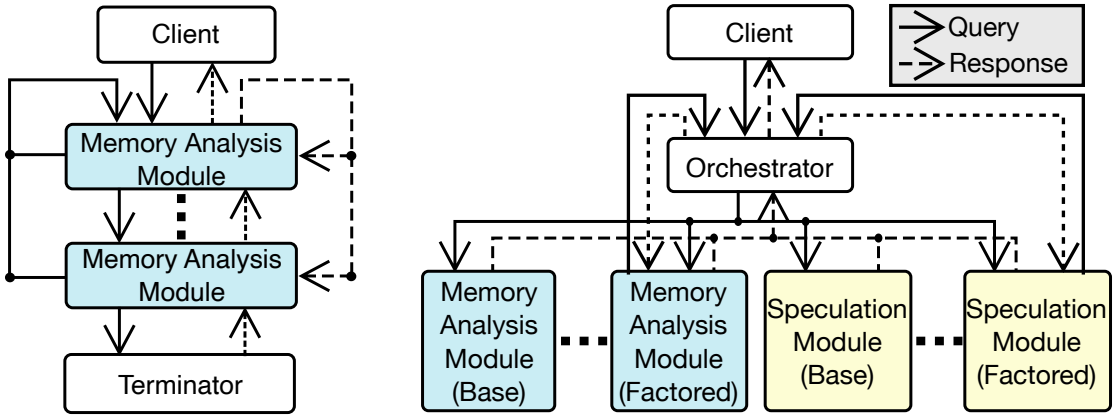
To maximize the impact of the control flow assertion (branch is never taken), interaction among control speculation and memory analysis is necessary in this example.

The *monolithic integration* approach would extend the kill-flow analysis algorithm [43] to interpret edge profiling information. This way, kill-flow can leverage the biased branch in the example, view $i1$ as executing on every iteration, infer that the condition *no-kill* from §2.1.1 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 §5.2.5.

5.2 Design

SCAF can be seen as a speculation-aware extension of CAF [43]. CAF is limited to collaboration among memory analysis algorithms, depicted as memory analysis modules in Figure 5.2. SCAF introduces speculation into the analysis framework with the introduction of *speculation modules* (Figure 5.2b). Speculation modules express the effect of speculative techniques by interpreting profiling information in terms of dependence analysis.



(a) CAF [43] (b) SCAF (This Work)

Figure 5.2: Design of Collaborative Analysis Frameworks

5.2.1 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 (same as in CAF [43]). 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 resolving the original queries. That way, modules are agnostic to who produces an incoming query or 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.

5.2.2 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 5.3 defines the syntax of the analysis queries (§5.2.2.1, §5.2.2.2) and the query responses (§5.2.2.3).

Query Syntax

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

Response Syntax

Query Response	r ::=	$(\mathcal{R}, \mathcal{S})$
Result	$\mathcal{R}$ ::=	$\mathcal{R}_a \mid \mathcal{R}_m$
Alias Result	$\mathcal{R}_a$ ::=	**NoAlias** | **MustAlias** | **SubAlias** | **MayAlias**
Modref Result	$\mathcal{R}_m$ ::=	**NoModRef** | **Ref** | **Mod** | **ModRef**
Set of Options	$\mathcal{S}$::=	$\emptyset \mid \{\mathcal{O}\} \mid \mathcal{S} + \mathcal{S} \mid \mathcal{S} \times \mathcal{S}$
Assertion Option	$\mathcal{O}$::=	$\emptyset \mid \{\mathcal{A}\} \mid \mathcal{O} + \mathcal{O}$
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 5.3: Syntax for SCAF's query and query response. Colored text indicates new syntax extensions over the query language of non-speculative analysis frameworks (CAF [43], LLVM [60]). `Before`, `After`, and `Same` denote that the first operation executes/the first pointer is computed in a strictly-earlier/a strictly-later/the same iteration than/as the second.

5.2.2.1 Query Types

As in LLVM's alias analysis infrastructure (LLVM 5.0 [60]) and CAF [43], 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.

5.2.2.2 New Query Parameters

This dissertation 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, this dissertation introduces optional *control-flow* query parameters in the form of dominator and post-dominator trees. That 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 for no-alias. Therefore, this dissertation 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 (§7.1.3) since modules can bail-out early if they cannot return the required answer.

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 the same static instruction may create several memory objects. Speculation analysis modules that reason about memory objects benefit from this context.

Moreover, queries in SCAF, same as in CAF [43], 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 parameter restricts the considered paths and allows distinguishability between intra-iteration (`Same`) and cross-iteration (`Before`, `After`) dependences.

5.2.2.3 (Speculative) Query Response

Memory analysis frameworks [60, 43] 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, SCAF's query responses 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 5.3). 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 §5.2.3 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.

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

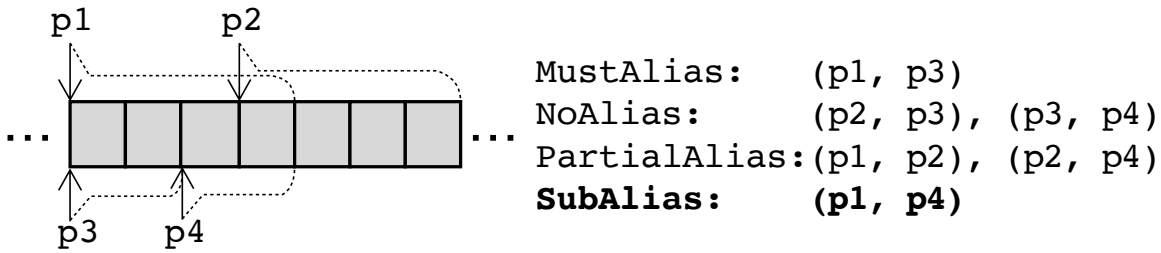


Figure 5.4: Difference between `MustAlias`, `NoAlias`, `PartialAlias`, and `SubAlias`. Arrows represent the pointed memory addresses, and dashed lines denote access sizes. Only the most precise result is presented. Analysis may return `MayAlias` when it cannot infer any other relation.

(iv) potential conflict points introduced by the application of this assertion. Clients use this information 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 §5.3.2.1 and on a per-module basis in §5.3.2.3 and §5.3.2.4.

Moreover, this dissertation 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` [60], where two memory objects are known to be overlapping in some way, but one is not necessarily contained within the other. Figure 5.4 (inspired by [61]) illustrates the differences among alias results. In this Figure, the memory location referenced by pointer `p4` is fully contained within the memory location referenced by `p1` but only overlaps with the memory location referenced by `p2`. `NoAlias` is inferred when there is no overlap, while pointers `p1` and `p3` have a `MustAlias` relationship since they point to the same memory address.

5.2.3 Orchestrator

The *Orchestrator* coordinates the interactions among modules and between modules and the client by forwarding queries to the modules and by processing query responses (Algo-

rithm 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)*

```
Input: Query
Output: 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;
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 [43, 60], 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 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

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

Input: Join Policy, Query Response 1, Query Response 2
Output: Combined Query Response
`/* Define assertion-related semantics */`
Def $\mathcal{O}_1 + \mathcal{O}_2 = \mathcal{O}_1 \cup \mathcal{O}_2$;
Def $\mathcal{S}_1 + \mathcal{S}_2 = \mathcal{S}_1 \cup \mathcal{S}_2$;
Def $\mathcal{S}_1 \times \mathcal{S}_2 = \{\mathcal{O}_1 + \mathcal{O}_2 : \mathcal{O}_1 \in \mathcal{S}_1, \mathcal{O}_2 \in \mathcal{S}_2\}$;
`/* Define order of precision of results */`
Def `pr(NoAlias) == pr(MustAlias) >pr(SubAlias) >pr(MayAlias);`
Def `pr(NoModRef) >pr(Mod) == pr(Ref) >pr(ModRef);`
$(\mathcal{R}_1, \mathcal{S}_1) \leftarrow$ r1;
$(\mathcal{R}_2, \mathcal{S}_2) \leftarrow$ r2;
if `pr`$(\mathcal{R}_1)$ `>pr`$(\mathcal{R}_2)$ **then return** *r1*;
if `pr`$(\mathcal{R}_1)$ `<pr`$(\mathcal{R}_2)$ **then return** *r2*;
`/* pr(`$\mathcal{R}_1$`) == pr(`$\mathcal{R}_2$`) */`
if $\mathcal{R}_1 == \mathcal{R}_2$ **then**
  switch *join_policy* **do**
    case *ALL* **return** $(\mathcal{R}_1, \mathcal{S}_1 + \mathcal{S}_2)$;
    case *CHEAPEST* **return** $(\mathcal{R}_1,$ `cheaper`$(\mathcal{S}_1, \mathcal{S}_2))$;
    case *Other Policies ...*;
`/* Special Case: Mod and Ref */`
else if $(\mathcal{R}_1 ==$ *Mod* **and** $\mathcal{R}_2 ==$ *Ref*$)$ **or** $(\mathcal{R}_1 ==$ *Ref* **and** $\mathcal{R}_2 ==$ *Mod*$)$ **then**
  if `conflict`$(\mathcal{S}_1, \mathcal{S}_2)$ **then**
    return `handleConflictingAssertions`*(r1, r2)*
  else
    return $($*NoModRef*$, \mathcal{S}_1 \times \mathcal{S}_2)$
else
  return `handleConflictingResults`*(r1, r2)*;

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.

For simplicity and lack of empirical evidence justifying exposure of all options and exhaustive search, this work's implementation opts 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.

5.2.4 SCAF within a Compiler

SCAF suggests: SCAF does *not* perform any transformation. SCAF merely makes *suggestions*. Clients can choose to ignore SCAF's suggestions to avoid paying the cost of their accompanying speculative assertions. Different configurations of the *Orchestrator* adjust the content of these suggestions, but 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 transformations 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.

5.2.5 Example

This section illustrates concepts and design decisions described in the previous sections with the motivating example from section §5.1.1. For this example (presented again in Figure 5.5a), a client wants to determine whether there is a cross-iteration dependence from instruction $i3$ to $i2$. 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). This dissertation proposes 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

```
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

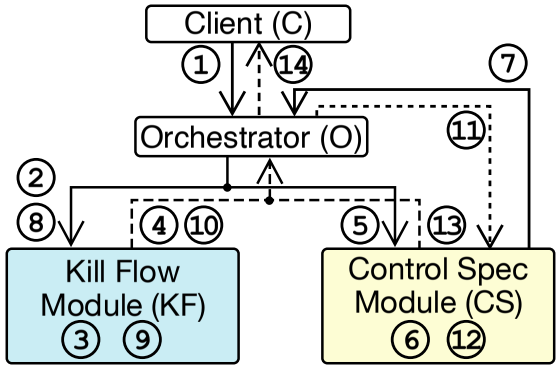
Figure 5.5: Motivating Code Example

in Figure 5.5b. 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 5.6 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 5.5c. 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 why this dependence did not manifest during profiling (i.e., not taken branch).



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 $\mathcal{O}$=$\{\mathcal{A}\}$, and response $\mathbf{r_3}$: (**NoModRef**, $\mathcal{O}$)
⑬	CS→O	Return $\mathbf{r_3}$
⑭	O→C	Return $\mathbf{r_3}$

Figure 5.6: A step-by-step example of SCAF in action. The client wants to determine if there is a cross-iteration data flow from i3 to i2 in the loop in Figure 5.5a. 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 either of these two modules. In Step 9, the kill-flow module perceives the code as in the *Speculative View* in Figure 5.5b due to the speculative control flow information (*spec_dt*, *spec_pdt*).

5.3 Implementation

SCAF is implemented on the LLVM Compiler Infrastructure [53] (version 5.0.2). This section describes the memory analysis (§5.3.1) and speculation (§5.3.2) modules included in SCAF's implementation.

5.3.1 Memory Analysis Modules

SCAF includes the 17 analysis algorithms described in CAF [43, 42]. Each of these algorithms tries to disprove one of the conditions described in §2.1.1. Some of these analyses reason about features of the LLVM IR and the C standard library (*Auto-Restrict*, *Basic Loop*, Φ-*Maze*, *Semi-Local*). The *Kill-Flow* and the *Callsite Depth-Combinator* modules look for flow-killing operations to disprove the *no-kill* condition in §2.1.1. Several memory analysis modules involve reachability algorithms, which reason about which object addresses can be stored in particular memory locations (*Global Malloc, Non-Captured Global, Non-Captured Source, Unique Access Paths*). Other algorithms disprove aliasing by reasoning about induction variables and the scalar evolution of pointers (*Array of Structures*, *SCEV*). The rest of the algorithms reason about shape analysis (*Sane Types, Non-Captured Fields, Acyclic, Disjoint Fields, Field Malloc*). Table 5.2 briefly summarizes the memory analysis modules implemented in SCAF.

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 §5.2.5.

Next, this section presents a critical improvement of this work to the Kill-Flow algorithm of CAF.

Table 5.2: Summary of memory analysis modules implemented in SCAF. It includes all the analysis algorithms described in CAF [43, 42]. An almost identical table is presented in [42]. For a more detailed description of these memory analysis algorithms, refer to Appendix A in [42].

Memory Analysis Module	Sensitivity				Demand-driven?	Premise Queries
	Memory-flow	Control-flow	Array/field	Calling-context		
Auto-restrict	×	×	×	✓	Partially	✓
Basic Loop	×	×	✓	×	Fully	×
Φ-Maze	×	×	×	×	Fully	×
Semi-Local	×	×	×	×	Partially	✓
Kill Flow	✓	✓	×	×	Fully	✓
Callsite	✓	✓	×	✓	Fully	✓
Global Malloc	✓	×	×	×	Partially	×
Non-Captured Global	×	×	×	×	Fully	×
Non-Captured Source	×	×	×	×	Fully	×
Unique Paths	✓	×	✓	✓	Partially	✓
Array of Structures	×	×	✓	×	Fully	✓
SCEV	×	×	×	×	Fully	×
Sane Types	✓	×	×	×	Partially	✓
Non-Captured Fields	✓	×	✓	×	Partially	✓
Acyclic	×	×	✓	×	Partially	×
Disjoint Fields	×	×	✓	×	Partially	×
Field Malloc	×	×	✓	×	Partially	✓

Extended Kill-Flow Analysis Algorithm: Kill-Flow is a highly effective analysis algorithm that searches for killing operations along all feasible paths between two operations. If a killing operation is found, then these two operations cannot have a dependence. Since there may be infinitely many paths, its search involves a timeout and is restricted to blocks that post-dominate the source of the queried dependence and dominate the destination. This approximation prevents the detection of a common pattern (seen in `052.alvinn`, `179.art`, `dijkstra`) that can be observed in the code in Figure 5.7. The write to **`pathcost`** in line **30** kills values flowing from the previous iteration to the read in line **42**. However, there is no dominance relation, and thus it cannot be detected. This work extends the Kill-Flow algorithm of prior work [43] to detect this pattern. Observe that the loop header of the inner loop in line **29** dominates the read in line **42**. The extended

```
21 void hot_loop(int N) {
26   for (src=0; src<N; src++) {
29     for (i=0; i<N; i++)
30       pathcost[i] = inf;
31
32     enqueue(src, 0);
33     while (!emptyQ()) {
34       int v = dequeue();
35       for (i=0; i<N; i++) {
39         nDist = adj[v][i] + dist;
42         if (pathcost[i] > nDist) {
45           pathcost[i] = nDist;
46           enqueue(i, nDist);
47         }
48       }
49     }
53   }
55 }
```

Figure 5.7: Shortened version of the dijkstra example from §4.4

Kill-Flow algorithm treats this inner loop as a single operation that overwrites a range of memory locations. This way, it can easily be proven that this range write overwrites at every outer-loop iteration the memory addresses read in line **42**. Therefore, this extension allows SCAF's Kill-Flow module to disprove additional data flows compared to the proposed algorithm in CAF, further reducing the need for memory speculation.

5.3.2 Speculation Modules

This section describes a design pattern for speculation modules (§5.3.2.1), enumerates the profilers that guide them (§5.3.2.2), and finally briefly describes the speculation modules implemented in SCAF (§5.3.2.3, §5.3.2.4). 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. Table 5.3 briefly summarizes the speculation modules implemented in SCAF.

5.3.2.1 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 [16, 32, 93]. To lower the validation cost, state-of-the-art compilers also implement less generic but cheaper-to-validate speculative transformations compared to memory speculation [44, 48, 93].

In this dissertation, 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 (§5.2.2) 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 comes from validation and recovery, as discussed in §2.3.1. This work models the validation cost but not the recovery cost. Estimation of the latter appears unnecessary given the speculative assertions used in SCAF. Validating speculative assertions' adds latency in the common case, not only in the case of misspeculation. Further, all speculative assertions in this work are high-confidence

(always hold true during profiling). Therefore, misspeculation is equivalently unlikely for all the assertions. The existence of less conservative speculation schemes with varying misspeculation rates would necessitate modeling 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 this implementation do not introduce any conflict points. For the rest, orthogonality in terms of coverage prevents conflicts. In particular, modules that modify the allocation sites of memory objects could conflict with each other, but the set of memory objects that such modules handle are disjoint, preventing any 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 §5.2.2 is respected. The 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: The proposed 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 modules 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 (§5.3.2.3), read-only (§5.3.2.4), and short-lived (§5.3.2.4) modules from separation speculation [44]) and participation of speculation modules in resolution of queries that go beyond their own reasoning (e.g., Figure 5.6).

5.3.2.2 Profilers

SCAF's speculation modules use information generated by a set of profilers: (i) an edge profiler that identifies biased branches [53]; (ii) a value-prediction profiler that detects predictable loads [33]; (iii) a pointer-to-object profiler that produces a points-to map, allowing detection of underlying objects for every memory access [44]; and, (iv) an object-lifetime profiler that detects short-lived memory objects, namely objects that exist only within a single loop iteration [44].

5.3.2.3 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

Table 5.3: Summary of speculation modules implemented in SCAF.

Speculation Module	Validation	Conflict Points	Demand-driven?	Premise Queries
Pointer-Residue Speculation	Few Bitwise & Branch Operations	None	Fully	×
Points-to Speculation	N/A	N/A	Fully	×
Control Speculation	Off-path Assertion (practically zero cost)	None	Partially	✓
Value Prediction	Branch Operation	None	Partially	✓
Read-only	Few Bitwise & Branch Operations	Modified Allocation Sites	Partially	✓
Short-lived	Few Bitwise, Arithmetic & Branch Operations	Modified Allocation Sites	Partially	✓

(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 [42].

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 it may return `SubAlias` (explained in §5.2.2.3). Validating points-to objects information is, in general, expensive and complicated. Thus, this module assigns 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 the read-only and short-lived modules (§5.3.2.4), 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.

5.3.2.4 Factored Speculation Modules

Same as factored memory analysis algorithms in CAF [43], 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 [43] (i.e., *Global Malloc, Non-Captured Global, Non-Captured 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, thus facilitating 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, the control

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

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. Thus, it does not introduce conflicts with other speculative assertions, as opposed to other schemes (e.g., [66]) 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 §2.1.1. 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. A `MustAlias` 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 [44]. 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. [44] 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 this work separates and decomposes the analysis part of separation speculation [44] to simple modules that collaboratively infer at least the same properties as the monolithic design proposed in [44]. 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 [44, 48, 93]. 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 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.

5.3.2.5 Recovery

Clients utilizing SCAF's query responses with speculative assertions need to insert the corresponding validation code (described in §5.3.2.3, §5.3.2.4) 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 [27, 46, 44, 77, 49] and thread-based [64, 93, 92, 94, 39] schemes. The implementation of the Per*spec*tive parallelizing compiler, which uses SCAF and is presented in §6, employs process-based runtime support for misspeculation recovery (§6.8).

Chapter 6

Parallelization Infrastructure Implementation

This chapter discusses implementation details for Per*spec*tive, a fully-automatic parallelization system that incorporates the ideas described in §4. Figure 6.1 depicts an overview of the Per*spec*tive framework, which includes a set of profilers, a parallelizing compiler, and a runtime system.

The compilation flow begins with a preprocessing step in which the code is canonicalized, and the profiling results are generated. In the planning phase, for each hot loop, the compiler queries the speculation-aware collaborative analysis framework (SCAF) to populate a program dependence graph (PDG) annotated with information utilized by the rest of the planning phase. Annotations include properties for the dependences and the dependent instructions. A memory speculation module provides additional annotations for dependences not manifested during profiling. The applicability guard of each enabling transformation examines the annotated PDG and creates transformation proposals that offer to remove parallelization-inhibiting cross-iteration dependences in the loop, along with their cost. Then, each applicable parallelization transformation produces a proposal that specifies the offered speedup along with the dependences that need to be removed for it to

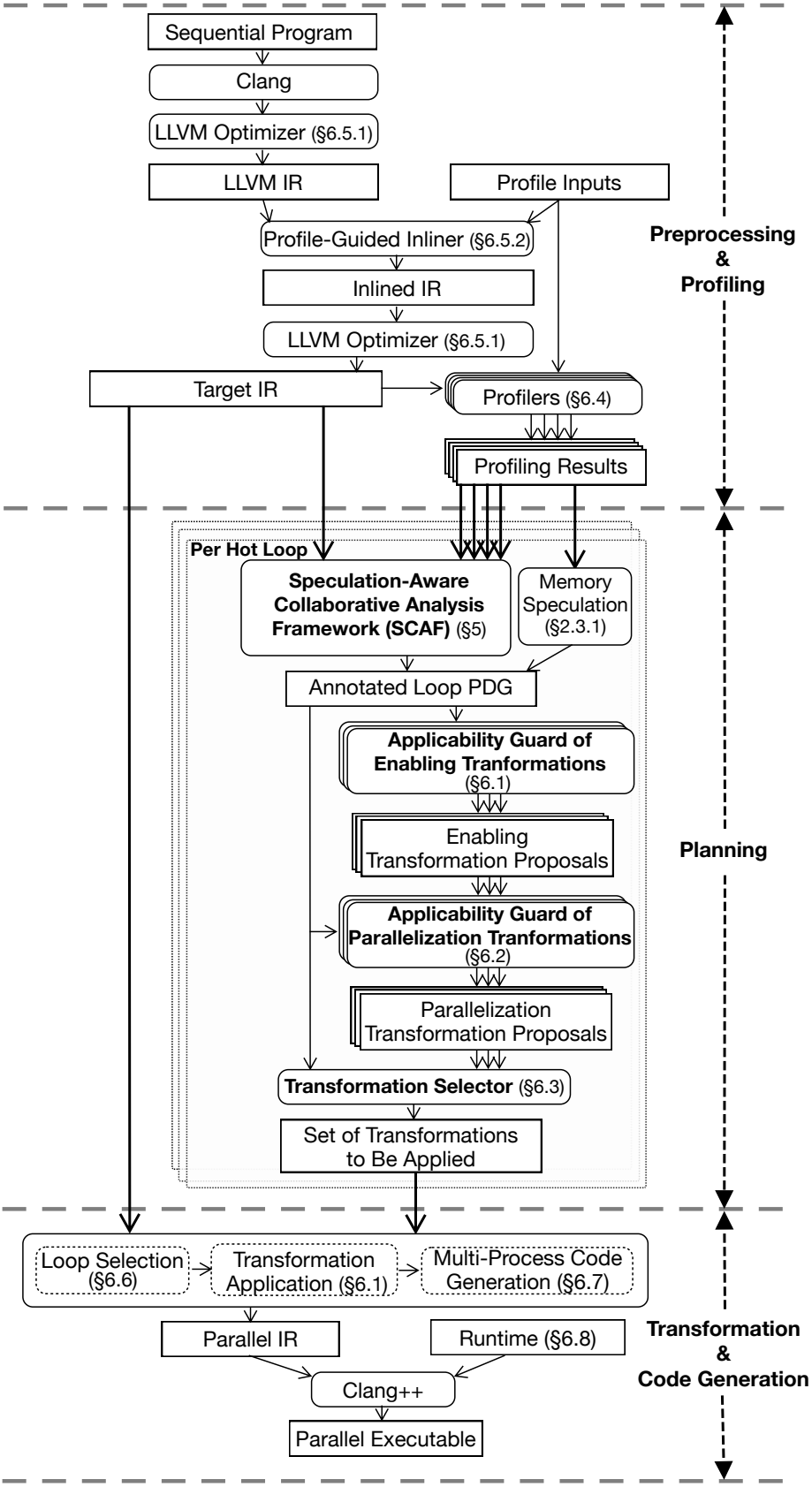


Figure 6.1: Per*spec*tive Framework Overview

be applicable. For DOALL parallelism, which is the focus of this dissertation, these are all the cross-iteration dependences. Next, the transformation selector picks for each parallelization proposal a minimal-cost set of enabling transformations that covers all target dependences, and then it selects the most profitable parallelization plan (if any). Finally, the compiler selects a set of compatible parallelizable loops with the maximum profitability, applies the transformations in their plans, and generates the parallel IR, which is then linked with the runtime and compiled to a parallel executable.

The rest of this chapter begins with the description of components (except for SCAF) in the planning phase, which includes the main contributions of this dissertation, and subsequently describes other parts of the framework. A detailed description of the design and the implementation of SCAF is provided in §5.

6.1 Enabling Transformations

Enabling transformations address memory, register or/and control cross-iteration dependences, which inhibit the parallelization of the target loops.

6.1.1 Memory Dependences

Applicability: A transformation's applicability guard uses the annotated PDG to find memory objects that satisfy a specific set of desired properties. For each found memory object, it records the needed speculative assertions. A transformation applicable for a memory object can handle all the cross-iteration memory dependences associated with this memory object.

Transformation Proposal: The output of each applicability guard is assembled in a transformation proposal that is sent to the transformation selector (§6.3). The proposal includes for each memory object an estimated handling cost based on the transformation

itself and the validation cost of all used speculative assertions. For simplicity, each type of transformation and speculation validation operation is assigned a fixed cost that ensures a basic ordering among the options. For example, memory speculation has an extremely high cost (expensive validation), loaded value prediction has a much smaller cost, while control speculation has no cost. For the set of transformations and speculative assertions in this implementation and in the context of DOALL parallelization, this simplified cost model proved sufficient to detect minimal-cost plans for the evaluated benchmarks.

Transformation Application: Each transformation reallocates memory objects it is selected to handle to its own heap, disjoint from any others; transformations may also perform additional transformation-specific modifications.

Separating objects is essential for two reasons. First, each transformation may demand different memory mapping semantics and handles objects differently at commit. Second, mapping of memory accesses to objects often relies on profiling information, especially in languages with unrestricted pointers like C/C++. Ensuring that all objects' accesses are contained within a transformation's heap is sufficient to validate underlying object assertions. This idea of heap separation has been explored previously by Johnson et al. (Privateer [44]).

Memory-related enabling transformations include:

- Privatization: Applicable for objects with no cross-iteration flow dependences (§2.3.2). Requires costly logging and merging at commit (§4.3).
- Reduction: Applicable for objects that only participate in reduction operations (§2.3.2). At commit, objects are merged according to their reduction operation.
- Short-lived: Applicable for objects that only exist within one iteration of the loop. Inserts check to ensure that all these objects are freed at the end of each iteration. Relies on the *short-lived* speculative assertions (§5.3.2.4).

- Read-only: Applicable for objects that are unmodified within the loop. Requires no transformation-specific checks. Relies on the *read-only* speculative assertions (§5.3.2.4).

- I/O deferral: Applicable for shared I/O objects (e.g., `stdout`). When applied, it replaces I/O library calls with custom calls. During runtime, it collects output operations and performs them in-order at commit.

Efficient Privatization Variants: This dissertation introduces four efficient variants of the privatization transformation. Their applicability criteria are described in §4.3.

- Independent: This transformation's heap is shared among all parallel workers since there are no overlapping memory accesses. No monitoring of write sets is needed. At commit, if the loop is speculatively parallelized, the heap is copied out to the non-speculative state.

- Overwrite Private: This transformation's heap has CoW (copy-on-write) mapping. At the end of the parallel invocation, the last executed iteration state is copied-out, and no monitoring is needed.

- Predictable Private: This transformation's heap has CoW mapping. The live-out state is predictable, so no monitoring or merging of parallel worker state is needed.

- Local Private: This transformation's heap has CoW mapping, and there is no need for copy-out or monitoring.

6.1.2 Register & Control Dependences

Cross-iteration register dependences are handled with reduction, replication, control speculation, or value prediction. Replication replicates side-effect-free computation across parallel workers to overcome cross-iteration dependences and avoid inter-thread communication. Cross-iteration control dependences are handled either with replication or control

speculation. The use of replication allows handling of uncounted loops, namely loops with unknown trip count when the loop is invoked. In terms of transformation cost, all transformations have constant costs except for replication. Replication's cost depends on how many instructions need to be replicated. Non-speculative enablers (reduction and replication) are preferred, in most cases, over speculative ones, and reduction is preferred over replication.

6.2 Parallelization Transformations

Parallelization transformations partition the code into work units that can execute in parallel. Multiple parallelization schemes have been proposed with varying degrees of effectiveness and applicability (§2.4). This dissertation focuses on the DOALL transformation, which is the most commonly used parallelization scheme. Yet, the planning phase of Per*spec*tive was built to generalize beyond DOALL. Although not evaluated in this dissertation,[1] the infrastructure[2] supports another parallelization transformation, the PS-DSWP [78] transformation described in §2.4.2.

Similarly to the enabling transformations, parallelization transformations are split into two parts: an applicability guard that produces parallelization proposals in the planning phase of the compiler, and the actual transformation that is applied (if selected) in the code generation phase of the compiler. The latter part is briefly discussed in §6.7.

The applicability guard of each parallelization transformation takes as inputs the annotated loop PDG and the proposals of the enabling transformations. These inputs combined essentially provide a simplified view of the dependence graph of the target loop, namely a loop PDG that only includes the non-removable dependences of the loop. Based on these inputs, the applicability guard determines whether the parallelization pattern in question

[1] DOALL parallelization was selected instead of PS-DSWP for all the evaluated benchmarks.

[2] The Perspective Parallelization Framework is available at: https://github.com/PrincetonUniversity/cpf.

is applicable for the target loop. If it is applicable, the guard produces a parallelization proposal.

The parallelization proposal includes an estimated loop speedup that the parallelization scheme can offer for the target loop. For example, DOALL parallelization, if applicable, can offer close-to-linear speedups. PS-DSWP's offered speedups mainly depend on the portion of the loop that is parallelizable. This speedup estimate does not incorporate the cost of applying any necessary enabling transformations. To avoid the application of unnecessary enabling transformations, the guard additionally includes in the proposal all the dependences that need to be removed for the transformation to be applicable (i.e., limiting dependences). For DOALL, these are all the cross-iteration dependences in the loop. PS-DSWP, after producing a pipeline based on the simplified PDG, reports as limiting dependences all the cross-iteration dependences in the parallel stages of the pipeline and all the dependences from later pipeline stages to earlier stages. Finally, the proposal also includes all the enabling transformation proposals that address at least one of these limiting dependences. The selection of the enabling transformations that will cover all the limiting dependences is left to the *transformation selector*.

6.3 Transformation Selector

The selector's goal is to pick the most efficient parallelization proposal. This selection is a two-step process. First, for each parallelization transformation proposal, the selector picks the cheapest set of enabling transformations that enables the proposed parallelization transformation. In the current implementation, it greedily selects the cheapest enabling transformation proposal for each limiting dependence reported in the parallelization transformation proposal. The second step of the selection involves picking the cheapest parallelization proposal taking into account both the reported speedup and the cost of the selected enabling transformations. In practice, the selector's implementation in Per*spec*tive always selects,

if available, the DOALL parallelization plan. Though one could always increase the complexity of this selection process, no empirical evidence justifies such extra complexity. Note that if no speculative assertions are used, the selector produces a non-speculative plan, forgoing the need for any speculation overhead. Naturally, if no parallelization proposal is provided, the selector reports that the loop is not parallelizable.

6.4 Profiling

Per*spec*tive employs all the profilers used by SCAF (§5.3.2.2), namely an edge profiler [53], a value-prediction profiler [33], a pointer-to-object profiler [44], and an object-lifetime profiler [44]. Additionally, Per*spec*tive uses a memory dependence profiler [18] (reports memory dependences observed during profiling) to fuel the memory speculation module.

6.5 Preprocessing

The compilation process begins with a preprocessing step that generates the targeted-for-parallelization intermediate representation (IR) of the program. The build system uses Clang [53] to generate LLVM IR from the sequential C/C++ programs, followed by LLVM IR optimizations. It then performs a pass of selective profile-guided inlining and finally another round of LLVM IR optimizations to produce the target LLVM IR that is used as the starting point for the rest of the compilation.

6.5.1 LLVM Optimizations

Transformations in this preprocessing step are crucial for the applicability and profitability of parallelization. Parallelizing compilers usually compile the source code with the *-O3* flag to get the initial IR and then perform a few additional passes. However, traditional compiler transformations are meant for optimized sequential execution. Some of these op-

timizations could unnecessarily complicate the code and preclude parallelization efforts. Any performance improvements from these optimizations are negligible compared to the benefits of successful parallelization. For example, LLVM tries to sink common instructions from two different execution paths. This reduces the code size, but when applied to memory operations, it complicates the inference of the underlying objects. To avoid such problems, Per*spec*tive's preprocessing step only applies a small selective set of LLVM IR enabling transformations that simplify and canonicalize the IR.

6.5.2 Profile-Guided Selective Inlining

Dependences involving callsites often prevent parallelization or lead to extensive use of expensive-to-validate memory flow speculation. Inlining can mitigate this problem, but the heuristics used to determine whether to inline or not in industrial compilers are tailored for sequential code optimization and are mostly irrelevant to effective parallelization.

Per*spec*tive uses profile information to detect hot loops and speculatively dead callsites. Only callsites that are within these hot loops and that cannot be speculated away with control speculation are inlined. Of these callsites, Per*spec*tive also avoids inlining ones that do not sink or source cross-iteration dependences, that inhibit parallelization.

6.6 Loop Selection

An execution time profiler – similar to gprof [90] but focused on loops – finds hot loops that execute for at least 10% of the total program execution. Out of the profitably parallelizable loops, certain loops are not selected for parallelization. The excluded loops are either simultaneously active with another more profitable loop (no support for nested parallelism), or their memory object heap assignments conflict with the assignments of a more profitable loop (each memory object can be allocated to only one heap in the current implementation).

6.7 Multi-Process Code Generation

Per*spec*tive's code generation (MPCG) is based on the multi-threaded code generation algorithm proposed by Ottoni et al. [71, 72]. MPCG generates efficient parallel code for both DOALL and PS-DSWP plans. In particular, MPCG takes as input for each target loop a parallelization plan (i.e., partitioning of the loop body into separate work units) and produces for each partition code that involves its corresponding instructions and additional communication instructions to preserve the original code's control and data flow. MPCG also generates code that invokes the runtime just before the loop invocation to set up the parallel workers and initiate the parallel execution.

6.8 Runtime

Per*spec*tive includes an efficient runtime for both speculatively and non-speculatively parallelized programs.

Process-based Approach: Per*spec*tive's runtime system uses a process-based parallelization scheme, as opposed to a thread-based one for multiple reasons. First, it allows the use of the copy-on-write (CoW) semantics of processes to communicate with low overhead live-in values from the main process to the workers. When speculation is used, it also offers an implicit separation between the speculative states of the workers and the committed state that the main process maintains. The benefits of process-based parallelization have also been discussed by prior work [27, 77, 44]. To enable cheap heap assignment validation, each worker's virtual memory address space is segmented into disjoint sections, corresponding to each transform's heap, same as in Privateer [44]. To avoid the overhead of process spawning for parallelized inner loops with multiple loop invocations (e.g., `052.alvinn`), each worker is spawned only once at program startup, and each worker's virtual memory is remapped at the start of every invocation.

Use of Shared Memory: The runtime system utilizes POSIX named shared memory to share data among workers and the main process. For non-speculative parallelization plans, the *independent* privatization's heap (§6.1.1) uses `mmap()` with shared permissions to avoid the overhead of merging worker states and copying out live-out values.

Checkpoints and Validation: Checkpoints are used to validate speculative memory accesses across workers and to save the current program state if no misspeculation is detected. Instead of using a separate validator thread [39, 77], Per*spec*tive employs a decentralized validation system, as in Privateer. When a worker reaches an iteration marked with a checkpoint operation, it acquires a lock to a shared checkpoint object, maps the checkpoint object to its virtual memory space to detect disallowed overlap with other workers, and then adds its own memory state to the object. If all workers complete the same checkpoint without misspeculation, the checkpoint object is committed to the non-speculative state maintained by the main process.

Recovery: The use of speculation necessitates recovery code in case misspeculation occurs. When misspeculation is detected by a worker, either during an iteration or at checkpoint time, other workers continue up to and commit the last valid checkpoint, then wait for recovery to finish. The main process will execute the loop sequentially up to and including the misspeculated iteration, using the last committed checkpoint as the starting state, and then restart the parallel workers.

Chapter 7

Evaluation

This chapter presents an experimental evaluation of the proposed compiler frameworks. Section 7.1 empirically evaluates the claim that SCAF significantly reduces the need for expensive-to-validate memory speculation compared to the best prior speculation-aware dependence analysis technique. Additionally, it investigates the source of this improvement and evaluates SCAF's query latency. Section 7.2 empirically evaluates the claim that Per*spec*tive maintains the applicability of prior automatic DOALL-parallelization systems while improving their efficiency. Further, it analyzes the performance effect of each contribution of this dissertation and explores the impact of misspeculation.

SCAF and Per*spec*tive are evaluated on a commodity shared-memory machine with two 14-core Intel Xeon CPU E5-2697 v3 processors (28 cores total) running at 2.60GHz (turbo-boost disabled) with 768GB of memory. The operating system is 64-bit Ubuntu 16.04.5 LTS with GCC 5.5. The evaluated compiler frameworks are implemented on the LLVM Compiler Infrastructure (version 5.0.2) [53].

7.1 Speculation-Aware Collaborative Analysis Framework

Benchmark Selection: SCAF is evaluated against 16 C/C++ benchmarks from the SPEC suites (SPEC CPU 92/95/2000/2006/2017) [87]. Fortran SPEC benchmarks are excluded

due to lack of Fortran front-end support (Flang [59] not supported in LLVM 5.0). Other C/C++ SPEC benchmarks are excluded due to the limitations of the profilers' implementation. All profilers (§5.3.2.2) 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. SCAF is evaluated on hot loops because improvements in memory dependence analysis for hot loops are expected to be more beneficial to clients than other parts of the benchmarks.

Profiling Data: Profiling information is gathered using the *train* inputs from the SPEC benchmark suites.

Client: SCAF is evaluated with a Program Dependence Graph (PDG) client [30]. 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 done in §7.2.3 in the context of a paralleling compiler.

Metric: Same as in prior work [43], analysis precision is measured with the `%NoDep` metric. 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 perfor-

mance response that is tied to the specifics of the evaluated optimizing client and thus an indirect measure. Moreover, the selected metric is indicative of a performance impact for an optimizing client, since performance is highly correlated with the cost of memory dependence removal in hot loops for certain types of clients, such as parallelization techniques [32, 64, 93]. This is further corroborated by the performance results presented in §7.2.3.

Best Prior Approach: The positive effect of collaboration among speculation modules and between memory analysis and speculation modules is evaluated by comparing SCAF against the best prior approach that integrates speculation into dependence analysis: *composition by confluence* (§5.1). This approach resembles prior proposals [103, 48, 64, 98] 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 isolate this dissertation's contributions from those of prior work (CAF [43] supports collaboration among memory analysis modules), all the memory analysis modules are treated as one component within which collaboration is permitted. This component will be referred to as CAF in the remainder of this evaluation section. 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 [82, 94, 64, 44, 48]. However, memory speculation is also the most expensive-to-validate speculation technique. SCAF aims to reduce the gap between the dependence coverage of memory speculation and inexpensive speculation techniques to

enable more profitable speculative optimizations. For a detailed description of memory speculation refer to §2.3.1.

7.1.1 Benefit of Collaboration

Figure 7.1 compares SCAF, *composition by confluence*, memory speculation, and CAF [43] using the `%NoDep` metric for the PDG client. Since resolving a memory dependence within a frequently executed loop has typically a larger impact for optimizing clients to that within a less frequently executed loop, `%NoDep` is recorded at a loop granularity and weighted by the loop's profiled execution time. In particular, for each benchmark, the reported result, referred to as dependence coverage, is a weighted average of the `%NoDep` of its loops.

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 7.1 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.

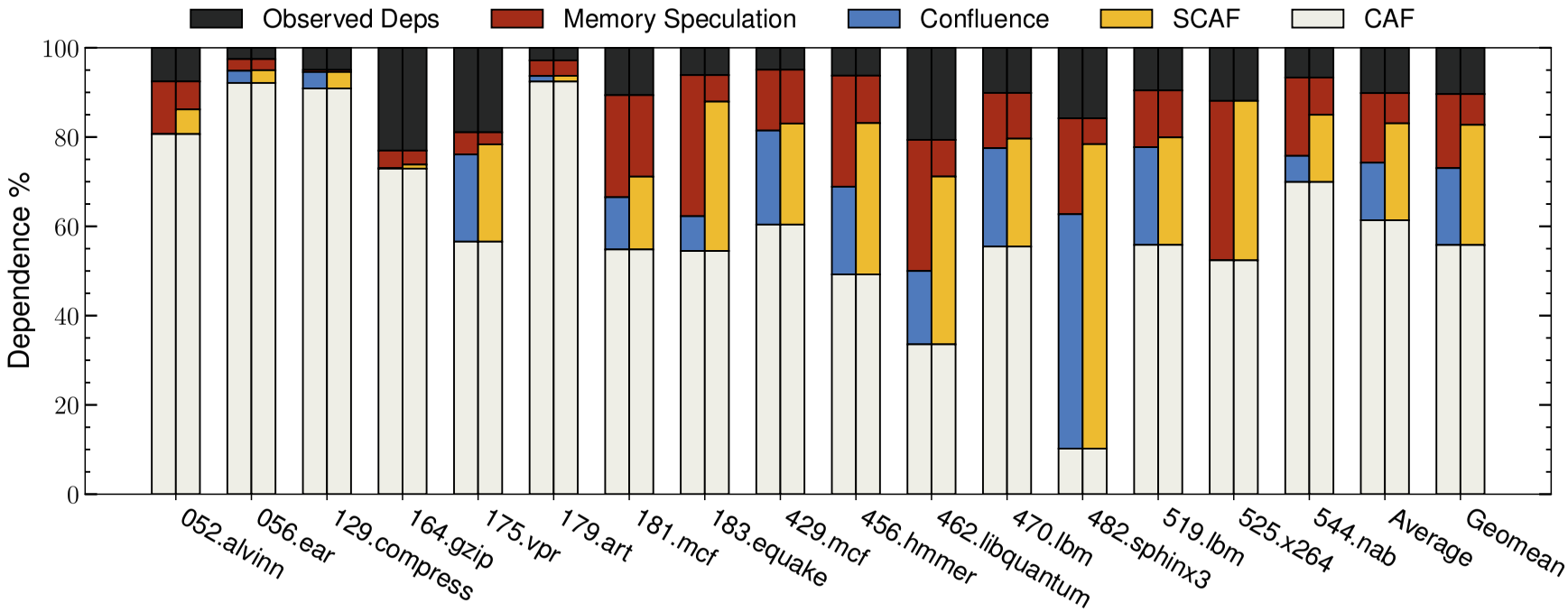


Figure 7.1: Dependence coverage by different schemes. *CAF* denotes dependences disproven by memory analysis (CAF [43]). *Confluence* and *SCAF* show additional dependences removed using inexpensive speculation without and with collaboration, respectively. *Memory speculation* asserts the absence of the remaining dependences that do not manifest during profiling. *Observed deps* are dependences that manifest during profiling.

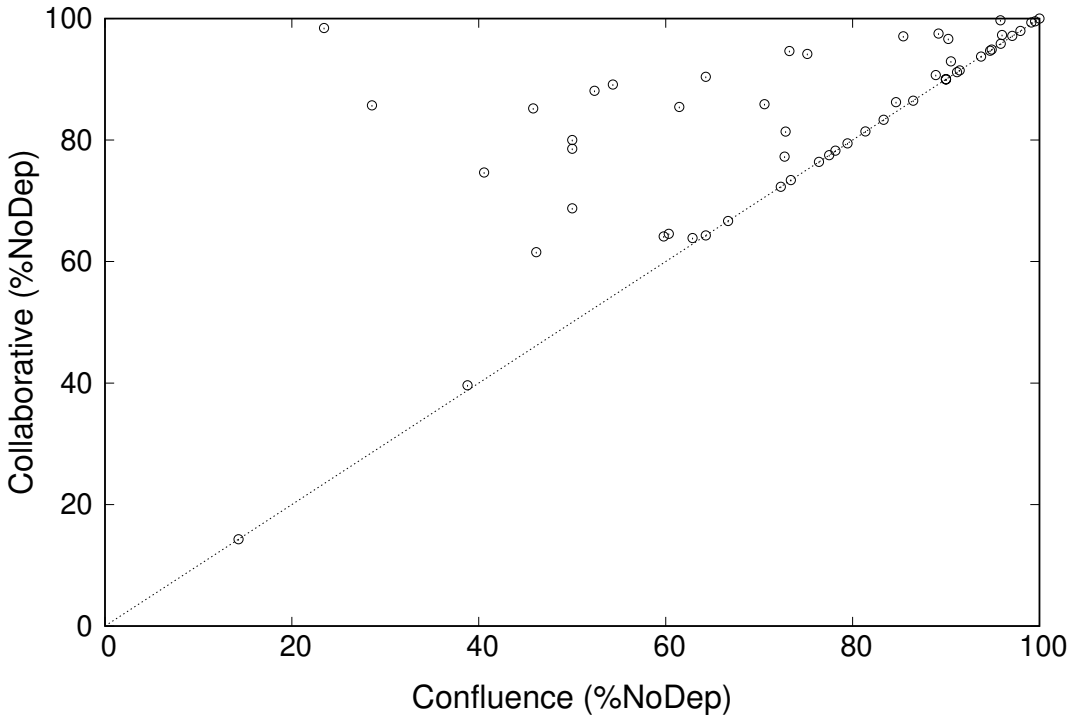


Figure 7.2: *Composition by Collaboration* (SCAF) compared with *Composition by Confluence*. Each point is a hot loop. Collaboration performs better on loops above the diagonal.

Figure 7.2 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 the 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`, and `179.art` benchmarks.

7.1.2 Contributions of Modules to Collaboration

This section evaluates which modules within SCAF participate in collaborations across the 16 evaluated benchmarks, contributing to the improvements in the `%NoDep` metric of

Table 7.1: Collaboration coverage of modules in SCAF across the 16 evaluated benchmarks 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

the PDG client (discussed in §7.1.1). Collaboration exhibits when two or more modules achieve higher precision than the confluence of their individual results.

Table 7.1 presents each module's contribution to collaboration. All the memory analysis modules are treated as one single component (CAF), since the focus is 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 resolving certain queries because the sum of the percentages of all the analysis modules for queries benefited by collaboration is bigger than 200%.

7.1.3 Query Latency

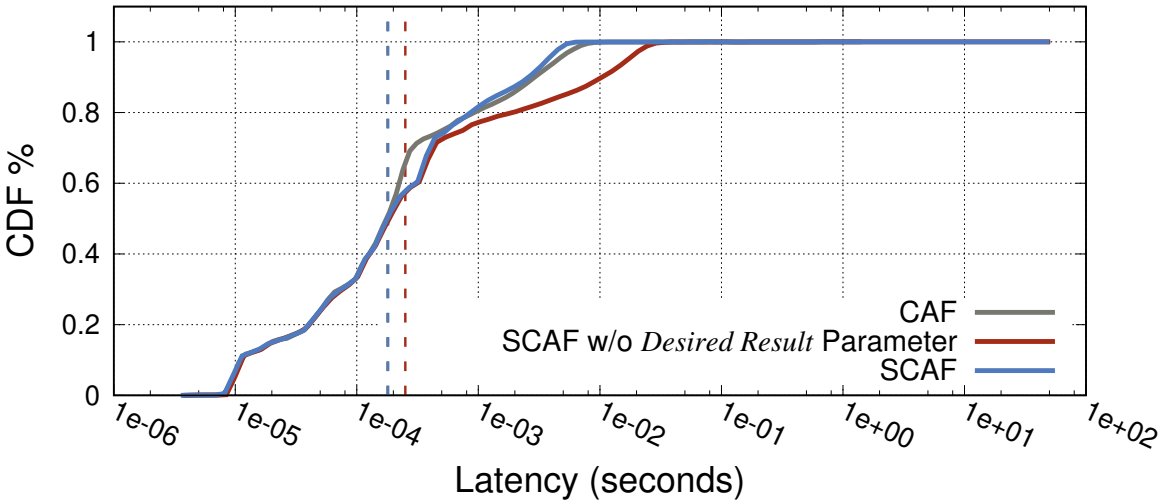


Figure 7.3: CDF of query latency for CAF [43], SCAF without the *Desired Result* parameter, and SCAF. The vertical colored dashed lines represent the geomean of each. The geomeans of SCAF and CAF are overlapping.

Figure 7.3 presents the cumulative distribution function (CDF) of the query latency for CAF [43], SCAF without *Desired Result* parameter (see §5.2.2.2 and Figure 5.3), and SCAF. All the queries performed by the PDG client are considered, and time is measured in processor cycles. 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.

7.2 Per*spec*tive Parallelization Framework

Per*spec*tive aims to boost the efficiency of automatic DOALL parallelization while maintaining the applicability of prior work. To validate this claim and enable credible and direct comparison with prior work, Per*spec*tive is evaluated against benchmarks that have been parallelized in prior work. In particular, Per*spec*tive is evaluated against 12 C and C++ benchmarks (Table 7.2), covering all the parallelizable (exhibiting speedup) benchmarks from two state-of-the-art automatic speculative-DOALL parallelization papers (Privateer [44], Cluster Spec-DOALL [48]) as well as an additional benchmark (179.art) from HELIX [15], a non-speculative automatic parallelization system.

The benchmarks from Polybench and the `dijkstra` benchmark from MiBench are modified to dynamically allocate previously statically allocated arrays and accept command line defined array sizes in the same way as prior work [44, 48]. Benchmarks are profiled using small inputs, while all the experiments presented in this section are conducted using different, large evaluation inputs. The evaluation inputs are chosen to be large enough for the sequential version to run for at least 10 minutes to observe accurate parallel execution times on 28 cores. For most of the evaluated benchmarks, the input arguments specify the problem size (e.g., size of a matrix) without any input data files. For the rest of the benchmarks, input data files included with the distribution of these benchmarks were used. Reported speedups are an average of 5 runs to minimize, although very small, the effect of variance in execution time between runs.

7.2.1 Scalability of Per*spec*tive

Figure 7.4 presents fully automatic whole program speedups across a various number of cores (up to 28 cores) for the 12 evaluated C/C++ benchmarks on a 28-core shared-memory commodity machine. These speedups are relative to the sequential performance of the original code, compiled with `clang++ -O3`. Per*spec*tive achieves scalable performance

Benchmark	Suite	% of Execution Time (Theoretical Speedup) [A]	SCAF's Crosss-Iter Dependence Cov [B]		New Enablers' Object Cov [C]	Monitored Read Set Size [D]				Monitored Write Set Size [D]			
			RAW	WAW		Privateer	v1	v2	Per*spec*tive	Privateer	v1	v2	Per*spec*tive
enc-md5	Trimaran	100.0% (28.0×)	**87**	**45**	**5**	1.87TB	9.21MB	39.1KB	39.1KB	581GB	581GB	43.2KB	43.2KB
052.alvinn	SPEC FP	97.5% (16.7×)	0	0	**4**	153GB	0B	0B	0B	107GB	59.9GB	4.08GB	10.2MB
179.art	SPEC FP	99.1% (22.5×)	**8**	**8**	**7**	1.6TB	64.8GB	64.8GB	0B	958GB	958GB	1.68GB	1.68GB
2mm	PolyBench	100.0% (28.0×)	N/A	N/A	**2**	1TB	0B	0B	0B	1TB	1GB	0B	0B
3mm	PolyBench	100.0% (28.0×)	N/A	N/A	**3**	3TB	0B	0B	0B	1.5TB	2.25GB	0B	0B
correlation	PolyBench	99.7% (25.9×)	N/A	0	0	0B	0B	0B	0B	192MB	192MB	192MB	192MB
covariance	PolyBench	99.9% (27.3×)	N/A	0	0	0B	0B	0B	0B	192GB	192MB	192MB	192MB
doitgen	PolyBench	99.6% (25.3×)	N/A	N/A	**2**	2.53TB	0B	0B	0B	2.54TB	10.1GB	0B	0B
gemm	PolyBench	100.0% (28.0×)	N/A	N/A	**1**	128MB	0B	0B	0B	256MB	256MB	0B	0B
blackscholes	PARSEC	99.7% (25.9×)	0	**1**	**1**	0B	0B	0B	0B	37.3GB	37.3GB	336B	336B
swaptions	PARSEC	100.0% (28.0×)	0	0	0	703KB	0B	0B	0B	165KB	165KB	165KB	165KB
dijkstra	MiBench	99.7% (25.9×)	**4**	**18**	**8**	973GB	648GB	648GB	0B	649GB	649GB	663MB	3.61KB

Table 7.2: Benchmark Details: (A) % of program execution time spent inside parallelized loop(s). The theoretical speedup is calculated using Amdahl's Law for 28 workers. (B) # of cross-iteration dependences that would require memory speculation without speculation awareness in memory analysis. "N/A" indicates all dependences are handled by static analysis. (C) # of objects covered by the proposed speculative privatization transformations. (D) Monitored read and write set sizes for speculation validation and privatization by Privateer and variants of Per*spec*tive; v1 represents Per*spec*tive without proposed enablers and SCAF (planning only); v2 represents Per*spec*tive without SCAF.

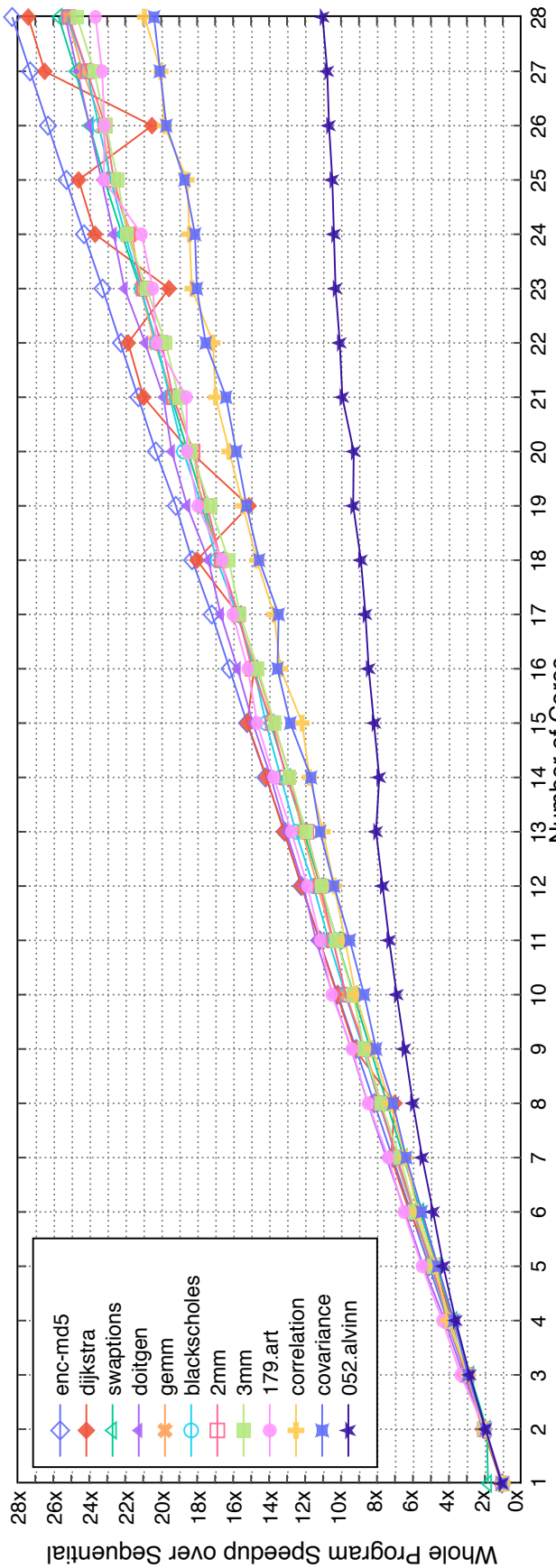


Figure 7.4: Per*spective*'s Fully Automatic Whole Program Speedup over Sequential Execution

on all the benchmarks thanks to the elimination of unnecessary overheads with the careful selection of applied transformations, the use of the speculation-aware collaborative analysis framework (SCAF), and the introduction of new enabling transformations.

For most of the benchmarks parallelized with Per*spec*tive, checkpointing does not add any significant (>1%) overhead; the exceptions are `052.alvinn`, `correlation`, and `covariance`, which exhibit lower-than-expected speedups. For `052.alvinn`, checkpointing constitutes a considerable portion of the run time (∼20%) since loop iterations are short, and thus the useful work performed between checkpoints is small. For `covariance` and `correlation`, the use of the basic privatization transformation entails that the checkpoints merge large private sets with an introduced overhead of ∼10% for each. Complex cross-iteration output dependences prevented the usage of more efficient privatization variants.

For several benchmarks, speedups exceed their theoretical limits, which is attributed to two factors: (1) The compiler replaces all calls to `malloc()` inside a parallelized loop with a custom heap allocator, same as in Privateer. This custom implementation does not track segments of memory that have been freed for later use in the way most C/C++ runtime libraries do, and as such, the overhead for dynamic (de)allocation is considerably reduced, as seen in `dijkstra`. (2) Using multiple cores increases the effective cache size, which may reduce access times to memory [41]. This effect can be seen in the performance of `179.art`, `enc-md5`, and `doitgen`.

7.2.2 Comparison with State-of-the-Art

Per*spec*tive is compared with Privateer [44], the most applicable prior automatic speculative-DOALL system. The authors of the Privateer paper provided the evaluated implementation of Privateer. This version was only modified to adhere to the more recent LLVM version used for Per*spec*tive. The observed speedups and runtime overheads of this implementation are similar to the results of the original paper.

Despite being a state-of-the-art parallelization framework, Privateer misses opportunities to reduce speculative checks and avoid monitoring of writes, as discussed in §3.2. This is apparent in columns (D) of Table 7.2, where parallelization of most benchmarks with Privateer requires monitoring of read and write sets orders of magnitude larger than those of Per*spec*tive. The first bar of Figure 7.5 corresponds to the achieved speedups by Privateer, and it demonstrates the performance impact of monitoring large read and write sets. These overheads are especially high for benchmarks with many reads and writes to privatized object(s) such as `3mm`, `doitgen`, `179.art`, and `dijkstra`. Overall, Per*spec*tive (last bar in Figure 7.5) doubles Privateer's geomean speedup by minimizing unnecessary memory access monitoring and checks.

7.2.3 Performance Analysis of Per*spec*tive

To quantify the impact of the three main contributions of this work (SCAF, planning, new enablers), two variants of Per*spec*tive with some components disabled were created. The first variant (*Planning Only / v1*) includes everything in Per*spec*tive except for SCAF and the new enabling transformations (i.e., efficient privatization variants in §6.1.1). Without SCAF, static analysis and speculation are utilized in isolation; a dependence cannot be resolved by a combination of static analysis and one or more speculative assertions. The second variant (*Planning & Proposed Enablers / v2*) is the same as *v1* but with the addition of efficient privatization variants (i.e., Per*spec*tive without SCAF). Figure 7.5 compares the performance of Per*spec*tive and Privateer with the two variants of Per*spec*tive.

The *Planning Only* variant of Per*spec*tive (*v1*) carefully selects the cheapest set of parallelization enabling transformations that need to be applied, as opposed to Privateer that aggressively applies speculative transformations. The introduction of this planning is the only distinguishing factor between Privateer and this variant; Privateer utilizes the same enablers and static analysis as this variant. Even so, this Per*spec*tive variant yields almost 46% geomean speedup over Privateer. The benefit of planning is particularly high for bench-

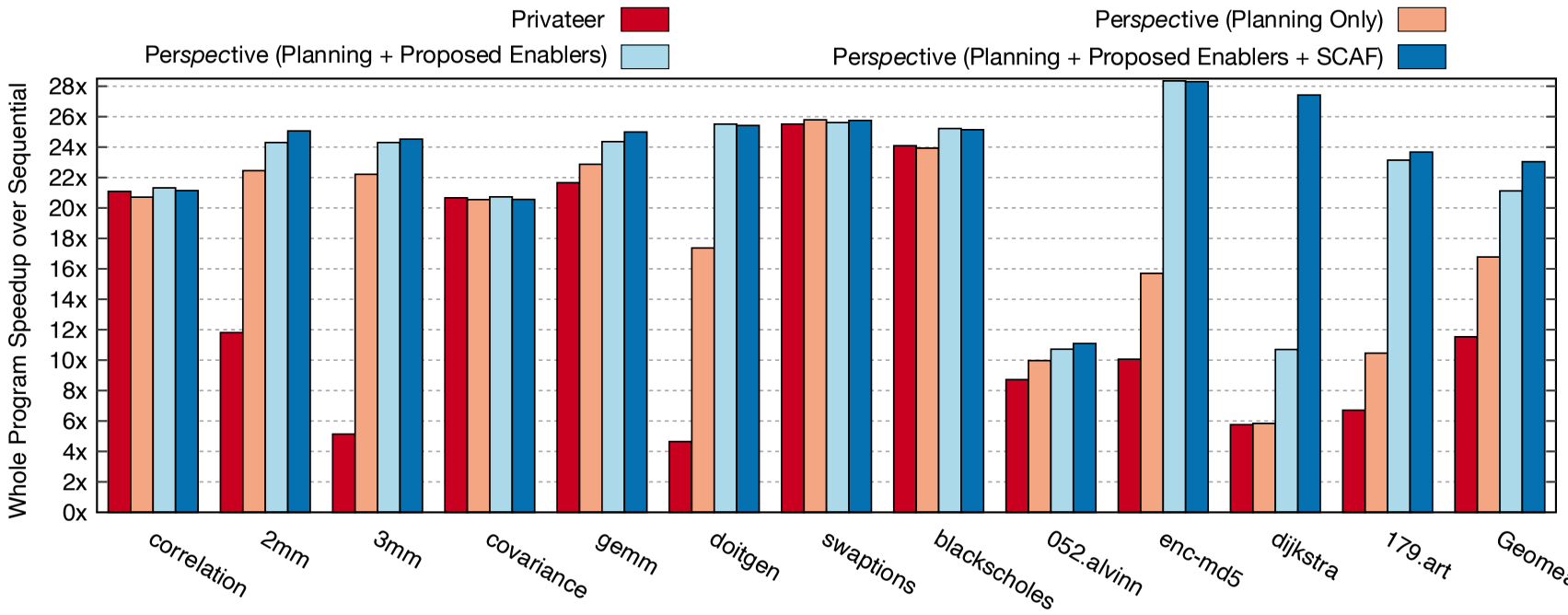


Figure 7.5: Whole Program Speedup Comparison among Privateer and Variants of Per*spec*tive with 28 Cores

marks with increased read set monitoring, including `2mm`, `3mm`, `doitgen`, `enc-md5`, and `179.art` (Figure 7.5). These performance improvements are mostly thanks to the avoidance of unnecessary checks on reads of non-speculatively privatized objects. Notice in columns (D) of Table 7.2 that for `doitgen` and `3mm`, the monitored read set size is reduced from 2.53TB and 3TB, respectively, to zero. The absence of instrumentation for certain reads additionally enables a peephole optimization that hoists monitoring of writes to the same location outside of the loop, further decreasing the overhead.

The introduction of new efficient privatization transformations (§4.3) in the *v2* variant improves the geomean performance over the *v1* variant by 26%, thanks to the avoidance of unnecessary bookkeeping. These new enablers are utilized in most of the evaluated benchmarks, as shown in column (C) of Table 7.2. The performance impact for each benchmark depends on the amount of avoided monitoring, depicted in columns (D). For example, the `179.art` and `dijkstra` benchmarks significantly benefit with the application of the *overwrite* privatization due to the dramatic reduction of the monitored writes. For the `enc-md5` benchmark, the use of *predictable* privatization contributes to the dramatic reduction of the monitored write set and the increased speedup, while the use of the *independent* privatization for the `2mm` and `3mm` benchmarks has a smaller performance impact. Note that just the introduction of these new enablers without the planning would not be effective. The planning phase is essential for exposing all the fine-grained information that makes these new transformations applicable and for allowing them to be selected over more expensive transformations.

The full version of Per*spec*tive additionally includes the speculation-aware collaborative analysis framework (SCAF, §5). With SCAF, static analysis by leveraging cheap-to-validate speculative assertions can remove additional dependences, thought by prior work to require expensive speculation, such as memory speculation. Its effect is seen most prominently in `dijkstra` (2.6× speedup over *v2*), where the use of control speculation in conjunction with static analysis enables efficient (without monitoring) privatization of

additional memory objects, including the global variable **`dist`** (discussed in §4.4). The introduction of SCAF also removes additional dependences from `179.art`, `enc-md5`, and `blackscholes`, as shown in columns (B) of Table 7.2. For the latter two, these removed dependences do not reduce the monitored memory accesses (columns (D)), and thus do not have any performance impact. For `179.art`, the monitored read set is nullified with the addition of SCAF, but the performance impact is small.

7.2.4 Misspeculation Evaluation

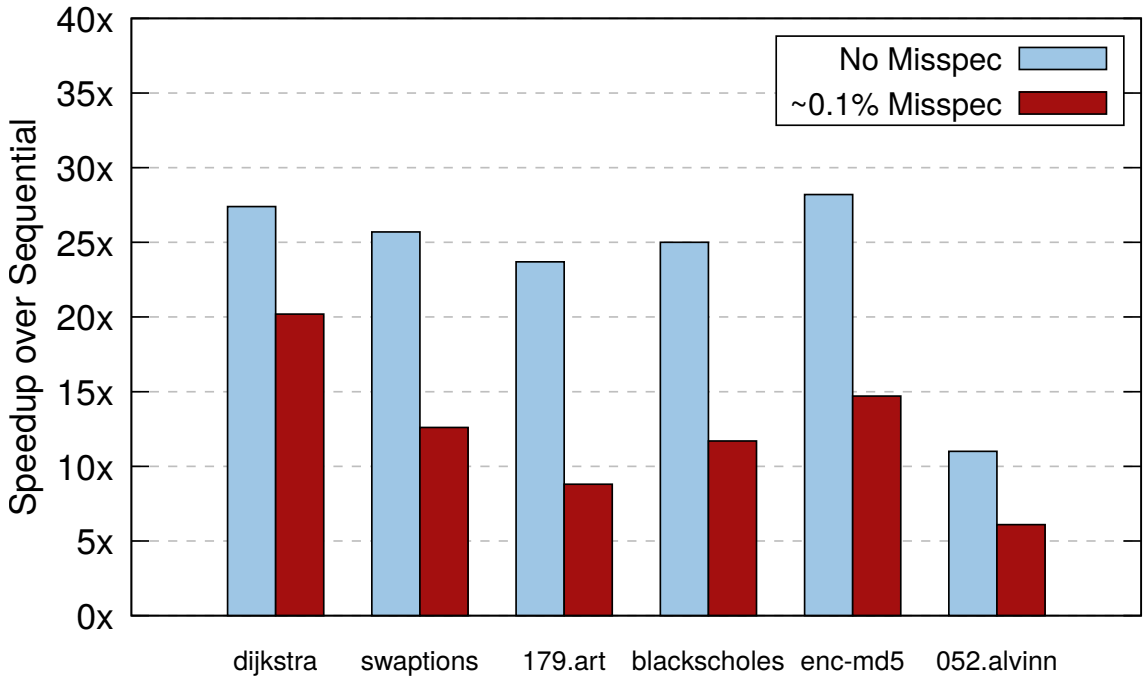


Figure 7.6: Impact of Misspeculation

Per*spec*tive uses only high-confidence speculation to minimize the chance of misspeculation. In particular, it only speculates properties that hold true without exception on the training inputs.[1]

This conservative approach led to a complete lack of misspeculation on the evaluation inputs for the eight speculatively parallelized benchmarks (`2mm`, `3mm`, `doitgen`, and

[1]The compiler is unaware of the evaluation inputs that are used for performance evaluation.

`gemm` were non-speculatively parallelized). Six of them could misspeculate for some (unusual) input. The remaining two (`covariance`, `correlation`) do not misspeculate across all possible inputs, since speculation is only used for heap separation checks that cannot fail (from manual inspection of the code). Even so, speculation is still necessary given the inability of static analysis to infer the underlying objects of certain memory accesses in these two C programs. Regardless of the accuracy of the used static analysis, such cases of non-misspeculating speculation cannot be completely eliminated due to the undecidability of static analysis [52].

Since none of the benchmarks exhibit misspeculation on the given inputs, misspeculation is artificially injected at the end of every 1000 iterations to observe the performance degradation with a misspeculation rate of 0.1%. The inputs for `179.art` were not large enough to allow for at least 1000 iterations. Therefore, a weighted average of non-misspeculating and misspeculating runs is performed to achieve an average corresponding to the desired rate.

Figure 7.6 shows how misspeculation affects the performance of the six benchmarks that could misspeculate for some input. These results demonstrate that misspeculation severely affects performance, and thus they support the decision to use only high-confidence speculation in Per*spec*tive.

Chapter 8

Related Work

8.1 Speculation-Aware Analysis

Johnson [42] proposes a design integrating speculation in a collaborative analysis framework (CAF [43]); 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 in [42] 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 [45], while SCAF is client-agnostic. SCAF specifies the required speculative assertions along with each query answer, allowing its clients to decide on how to act upon this information.

Other works [29, 26] 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.

Neelakantam et al. [66] propose converting biased branches to assertions to allow subsequent transformations to leverage speculative control flow information. Instead, SCAF leverages speculative control flow information during analysis and planning, prior to transformation.

In terms of static analysis, prior works [51, 20, 67, 11, 12, 56, 43] 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 [85] and sensitivity analysis [84] 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 [48, 98] 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. [58] 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. [62] 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.

8.2 Parallelizing Compilers

Early non-speculative DOALL parallelizing compilers (Polaris [10], SUIF [3, 100]) are limited by the imprecision of static analysis. LRPD [83] and R-LRPD [23] overcome the limitations of static analysis by leveraging speculation. Yet, these works are limited to array-based applications and cannot handle pointers and dynamic data structures.

More recent works (STMlite [64], CorD [93], Cluster Spec-DOALL [48], Privateer [44]) extend the applicability of automatic DOALL parallelization to general-purpose programs with profile-guided speculation. Per*spec*tive mitigates core inefficiencies of these prior works while maintaining their increased applicability.

Beyond DOALL, prior work has explored alternative parallelization paradigms (HELIX [15], DOACROSS [21], DSWP [71], PS-DSWP [78]) that tolerate more dependences than DOALL parallelization by allowing communication among workers. Per*spec*tive mainly targets DOALL parallelism, but it also involves extensions for PS-DSWP. In general, this dissertation's contributions are for the most part agnostic to the used parallelization scheme and thus should be profitable to a variety of parallelization paradigms. Evaluation of the impact of this work for parallelization schemes beyond DOALL is left for future work.

Other works [50, 74, 25, 75, 68] propose systems that require developers to cast programs in specialized code patterns or insert annotations to better express their intent. In-

stead, Per*spec*tive fully automatically parallelizes general-purpose applications without the need for annotations or specialized abstractions.

Another line of work [14, 97, 65, 95] extracts parallelism by ignoring data dependences without preserving soundness via misspeculation detection and recovery. These approaches extract parallelism either by sacrificing the program's output quality [14, 97, 65] or by relying on the user's approval [95]. Instead, Per*spec*tive extracts parallelism without violating the sequential program semantics.

8.3 Planning

The interaction of optimizing transformations can be quite complex. One transformation may enable, disable, or drastically change the behavior of subsequent transformations in unpredictable ways. Although many compilers ignore this problem by following a fixed order of phases that seems to work well in practice (e.g., `-O3` in `clang` or `gcc`), no fixed order of phases can be optimal [1, 19, 89, 101]. This is referred to as the *phase-order* problem [19]. Prior works have used human reasoning [101], machine learning [1, 7, 63], search heuristics [2], and novel intermediate representations [89] to explore the transformations' interactions and mitigate the phase-order problem. All of these approaches target ILP and yield small improvements.

The approach proposed in this dissertation bypasses the phase-order problem owing to two key insights. First, applicable parallelization transformations yield gains at a much higher order of magnitude than other compiler optimizations. Second, the issues limiting the applicability of these parallelization transformations and the effect of explored parallelization-enabling transformations are predictable and easily expressible on a PDG. Guided by these two insights, this work focuses on enabling parallelization transformations, simplifying the task of optimal selection of applied transformations.

Chapter 9

Conclusion and Future Directions

9.1 Conclusion

By identifying and mitigating core inefficiencies of prior speculative automatic parallelization systems, this dissertation represents an important advance in fulfilling the promise of automatic parallelization.

This dissertation presents 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.

This dissertation also presents Per*spec*tive, a parallelization framework that avoids overheads of prior work by combining SCAF with new efficient variants of speculative privatization and a new parallelizing-compiler design that enables careful planning. Per*spec*tive

fully-automatically yields a geomean whole-program speedup of 23.0× over sequential execution for 12 C/C++ benchmarks on a 28-core shared-memory commodity machine, doubling the performance of the state-of-the-art.

The source code of the compiler infrastructure built for this dissertation is publicly available.[1] Further, this dissertation is accompanied by publicly available artifacts that can be used to reproduce its evaluation results and corroborate its claims.[2]

9.2 Future Directions

9.2.1 Impact for Pipelined Parallelism

This dissertation evaluates its contributions in the context of DOALL parallelization. Evaluation for pipelined parallelism (§2.4.2) is left for future work. The infrastructure already supports PS-DSWP parallelization (§6.2). However, there is a lack of support for validation of certain speculative assertions for a PS-DSWP parallelization plan. The challenge is that PS-DSWP splits each loop iteration to different workers, contrary to DOALL parallelization where each loop iteration is executed entirely by a single worker. This necessitates adjustments in terms of speculation validation. For example, validating that a memory object is short-lived (i.e., exists only within one loop iteration) requires checking that the object is allocated and de-allocated within each iteration. This check is trivial and worker-local for DOALL, but for PS-DSWP this validation is non-trivial since an object might be allocated by one worker and freed by another worker. Fully leveraging this dissertation's contributions in the context of pipelined parallelism will considerably increase the applicability of the proposed compiler technology.

[1] SCAF is available at: https://github.com/PrincetonUniversity/SCAF. The Per*spec*tive Parallelization Framework is available at: https://github.com/PrincetonUniversity/cpf.

[2] The artifact for SCAF is available at: https://doi.org/10.5281/zenodo.3751586. The artifact for the Per*spec*tive Parallelization Framework is available at: https://doi.org/10.5281/zenodo.3606885. These artifacts have been awarded all top ACM reproducibility badges by the artifact evaluation committee of the PLDI '20 and ASPLOS '20 conferences, respectively.

9.2.2 Efficient and Robust Profiling

As discussed in §7.1, the implementation of profilers for the presented infrastructure lacks industrial-level robustness. This translates to frequent failures to produce profile data, blocking the evaluation of the proposed techniques across a broader range of programs. Further, the use of instrumentation-based profilers introduces excessive profiling overheads. More efficient and robust profilers are a necessary advancement for developing a robust speculative parallelization system. Inspired by advancements in industrial-grade profilers [17], a reasonable next step is to design sampling-based profilers that can accommodate the needs of advanced speculative parallelizing compilers with minimum overhead.

9.2.3 Broader Language Support

A compiler built on top of LLVM supports (by default) various source languages (e.g., C, C++, Rust, Fortran) thanks to available frontends that emit LLVM IR from programs in these languages. In practice, though, an optimizing compiler is not equally effective across all these languages. The compiler infrastructure presented in this dissertation is built and optimized for C programs (and secondarily for C++ programs). Better support for C++ would involve profile-guided devirtualization that would complement the currently-available static devirtualization that is insufficient for complex C++ SPEC benchmarks. Effective support of additional languages by the parallelizing compiler poses both engineering and research challenges. Engineering challenges include handling a variety of LLVM IR patterns produced by a diverse set of language constructs and understanding the behavior of each language's standard library functions (e.g., recognizing pure functions). Research challenges include leveraging rich language features that are lost when lowering the programs to LLVM IR. For example, Rust offers a strong type system that enables precise computation of memory dependences. Yet, the lowered code in LLVM IR is deprived of this critical type information. Therefore, using the existing compiler infrastructure to parallelize Rust programs will yield a sub-optimal result. Instead, one should either bring

this information into LLVM IR (e.g., using LLVM intrinsic functions) or move partially (or fully) the parallelizing compiler to a higher-level IR representation (e.g., MLIR [54]).

9.2.4 Beyond CPUs

With the end of Moore's law and with new emerging applications came extreme heterogeneity in hardware. Abstractions designed to hide hardware-specific characteristics are constantly broken today when scalability and efficiency are the goals. Programmers need to deal with new and complex programming models for each new parallel architecture. As a response to this problem, the ultimate goal is to develop parallelizing compilers that automatically target all these different parallel architectures (e.g., GPUs, spatial architectures). The ideas presented in this dissertation enable more efficient dependence handling, addressing fundamental automatic parallelization limitations that are not specific to multicore CPU-based systems. Therefore, this work may fuel new compiler advancements that enable automatic extraction of parallelism for other parallel architectures.

9.2.5 General-purpose Accelerators

The development of speculative, superscalar out-of-order processors is a premier success story in accelerating sequential codes. With the right compiler technology, propelled by the advancements presented in this dissertation, an analogous and complementary success story will unfold. This time, the story is one of accelerating sequential codes by exposing coarser-grained parallelism for execution over multiple cores. Routine extraction of thread-level parallelism from sequential codes enables the effective use of an abundance of cores, reducing the demands on the complexity and performance of individual cores. Latency hiding in sequential codes can occur among cores, not just within them. Simpler branch predictors, smaller caches per core, and in-order cores can mean more cores with the same number of transistors and more effective use of the silicon.

Bibliography

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[2] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 231–239, New York, NY, USA, 2004. ACM Press.

[3] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 126–138, Albuquerque, New Mexico, USA, June 1993. Association for Computing Machinery.

[4] Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language. Technical report, 1994.

[5] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I. August. 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, pages 351–367, Lausanne, Switzerland, March 2020. Association for Computing Machinery.

[6] Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I. August. SCAF: A Speculation-Aware Collaborative Dependence Analysis Framework. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 638–654, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Trans. Archit. Code Optim.*, 14(3):29:1–29:28, 2017.

[8] Utpal Banerjee. *Loop Parallelization*. Springer US, 1994.

[9] Sam Blackshear, Bor-Yuh Evan Chang, and Manu 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, Seattle, Washington, USA, June 2013. Association for Computing Machinery.

[10] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[11] Martin Bravenboer and Yannis 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, Chicago, IL, USA, July 2009. Association for Computing Machinery.

[12] Martin Bravenboer and Yannis 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, Orlando, Florida, USA, October 2009. Association for Computing Machinery.

[13] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 217–228. IEEE Press, 2014.

[14] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. Helix-up: Relaxing program semantics to unleash parallelization. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pages 235–245, Feb 2015.

[15] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, San Jose, California, March 2012. Association for Computing Machinery.

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

[17] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, New York, NY, USA, 2016.

[18] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. Data Dependence Profiling for Speculative Optimizations. In Evelyn Duesterwald, editor, *Compiler Construction*, Lecture Notes in Computer Science, pages 57–72, Berlin, Heidelberg, 2004. Springer.

[19] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: Adaptive compilation made efficient. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 69–77, New York, NY, USA, 2005. ACM.

[20] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The Reduced Product of Abstract Domains and the Combination of Decision Procedures. In Martin Hofmann, editor, *Foundations of Software Science and Computational Structures*, Lecture Notes in Computer Science, pages 456–472, Berlin, Heidelberg, 2011. Springer.

[21] R. Cytron. DOACROSS: Beyond Vectorization for Multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing (ICPP)*, pages 836–884, 1986.

[22] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[23] Francis H. Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, page 318, USA, April 2002. IEEE Computer Society.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, page 10, San Francisco, CA, December 2004. USENIX Association.

[25] Enrico A. Deiana, Vincent St-Amour, Peter A. Dinda, Nikos Hardavellas, and Simone Campanoni. Unconventional Parallelization of Nondeterministic Applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 432–447, Williamsburg, VA, USA, March 2018. Association for Computing Machinery.

[26] David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 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, pages 348–362, Williamsburg, VA, USA, March 2018. Association for Computing Machinery.

[27] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 223–234, San Diego, California, USA, June 2007. Association for Computing Machinery.

[28] P. Feautrier. Array expansion. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 99–111, New York, NY, USA, 1988. Association for Computing Machinery.

[29] Manel Fernández and Roger Espasa. Speculative alias analysis for executable code. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 222–231, Washington, DC, USA, 2002. IEEE Computer Society.

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

[31] Jordan Fix. *Hardware MultiThreaded Transactions: Enabling Speculative Multi-Threaded Pipeline Parallelization For Complex Programs*. PhD Thesis, Department of Computer Science, Princeton University, Princeton, NJ, United States, January 2020.

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

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

[34] Rakesh Ghiya and Laurie J. Hendren. 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, pages 1–15, St. Petersburg Beach, Florida, USA, January 1996. Association for Computing Machinery.

[35] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 256–265, San Diego, California, USA, June 2007. Association for Computing Machinery.

[36] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, USA, December 2001. IEEE Computer Society.

[37] Michael Hind. 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, pages 54–61, Snowbird, Utah, USA, June 2001. Association for Computing Machinery.

[38] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 146157, New York, NY, USA, 1988. Association for Computing Machinery.

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

[40] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 121–130, New York, NY, USA, 2010. Association for Computing Machinery.

[41] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: parallel speedup estimates for serial programs. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 519–536, Portland, Oregon, USA, October 2011. Association for Computing Machinery.

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

[43] Nick P. Johnson, Jordan Fix, Stephen R. Beard, Taewook Oh, Thomas B. Jablin, and David I. August. A collaborative dependence analysis framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 148–159, Austin, USA, February 2017. IEEE Press.

[44] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Speculative separation for privatization and reductions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 359–370, Beijing, China, June 2012. Association for Computing Machinery.

[45] Nick P. Johnson, Taewook Oh, Ayal Zaks, and David 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, Seattle, Washington, USA, June 2013. Association for Computing Machinery.

[46] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. 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, pages 157–168, USA, March 2009. IEEE Computer Society.

[47] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[48] Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. Automatic speculative DOALL for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 94–103, San Jose, California, March 2012. Association for Computing Machinery.

[49] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. Scalable Speculative Parallelization on Commodity Clusters. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 3–14, USA, December 2010. IEEE Computer Society.

[50] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, San Diego, California, USA, June 2007. Association for Computing Machinery.

[51] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher 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, Baltimore, Maryland, June 2005. Association for Computing Machinery.

[52] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[53] Chris Lattner and Vikram Adve. 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, page 75, Palo Alto, California, March 2004. IEEE Computer Society.

[54] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law, 2020.

[55] Chris Lattner, Andrew Lenharth, and Vikram Adve. 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, pages 278–289, San Diego, California, USA, June 2007. Association for Computing Machinery.

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

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

[58] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimiza-

tions. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 289–299, San Diego, California, USA, May 2003. Association for Computing Machinery.

[59] LLVM Project. Flang: A compiler front-end for Fortran. https://github.com/llvm/llvm-project/tree/master/flang, September 2020.

[60] LLVM Project. LLVM Alias Analysis Infrastructure. http://llvm.org/docs/AliasAnalysis.html, September 2020.

[61] Maroua Maalej and Laure Gonnord. Do we still need new Alias Analyses? report, Université Lyon Claude Bernard / Laboratoire d'Informatique du Paralllisme, November 2015.

[62] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. Generalized profile-guided iterator recognition. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 185–195, Vienna, Austria, February 2018. Association for Computing Machinery.

[63] Luiz G. A. Martins, Ricardo Nobre, João M. P. Cardoso, Alexandre C. B. Delbem, and Eduardo Marques. Clustering-based selection for the exploration of compiler optimization sequences. *ACM Trans. Archit. Code Optim.*, 13(1):8:1–8:28, 2016.

[64] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 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, pages 166–176, Dublin, Ireland, June 2009. Association for Computing Machinery.

[65] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Trans. Embed. Comput. Syst.*, 12(2s), May 2013.

[66] Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 174–185, San Diego, California, USA, June 2007. Association for Computing Machinery.

[67] Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[68] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic galois: on-demand, portable and parameterless. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, ASPLOS '14, pages 499–512, Salt Lake City, Utah, USA, February 2014. Association for Computing Machinery.

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

[70] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. October 2007.

[71] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Barcelona, Spain, November 2005. IEEE Computer Society.

[72] Guilherme de Lima Ottoni. *Global Instruction Scheduling for Multi-Threaded Architectures*. PhD Thesis, Department of Computer Science, Princeton University, Princeton, NJ, United States, September 2008.

[73] Manohar K. Prabhu and Kunle Olukotun. 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, pages 1–12, San Diego, California, USA, June 2003. Association for Computing Machinery.

[74] Prakash Prabhu, Stephen R. Beard, Sotiris Apostolakis, Ayal Zaks, and David I. August. Memodyn: Exploiting weakly consistent data structures for dynamic parallel memoization. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[75] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 1–11, New York, NY, USA, 2011. Association for Computing Machinery.

[76] William Pugh. 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, pages 4–13, Albuquerque, New Mexico, USA, August 1991. Association for Computing Machinery.

[77] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth International Conference on Architectural support for pro-*

gramming languages and operating systems, ASPLOS XV, pages 65–76, Pittsburgh, Pennsylvania, USA, March 2010. Association for Computing Machinery.

[78] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 114–123, Boston, MA, USA, April 2008. Association for Computing Machinery.

[79] Easwaran Raman, Neil Va hharajani, Ram Rangan, and David I. August. Spice: Speculative parallel iteration chunk execution. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 175–184, New York, NY, USA, 2008. Association for Computing Machinery.

[80] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 177–188, USA, 2004. IEEE Computer Society.

[81] Lawrence Rauchwerger and David Padua. The privatizing DOALL test: a run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 8th international conference on Supercomputing*, ICS '94, pages 33–43, Manchester, England, July 1994. Association for Computing Machinery.

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

[83] Lawrence Rauchwerger and David A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization.

IEEE Transactions on Parallel and Distributed Systems, 10(2):160–180, February 1999.

[84] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 263–273, Seattle, Washington, June 2007. Association for Computing Machinery.

[85] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. *International Journal of Parallel Programming*, 31(4):251–283, August 2003.

[86] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 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, pages 16–31, St. Petersburg Beach, Florida, USA, January 1996. Association for Computing Machinery.

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

[88] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, St. Petersburg Beach, Florida, USA, January 1996. Association for Computing Machinery.

[89] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, New York, NY, USA, 2009. ACM.

[90] The GNU Project. *GNU Binutils*. Published: \http://www.gnu.org/software/binutils/.

[91] The IEEE and the Open Group. *The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition*. 2004.

[92] Chen Tian, Min Feng, and Rajiv Gupta. Speculative parallelization using state separation and multiple value prediction. In *Proceedings of the 2010 international symposium on Memory management*, ISMM '10, pages 63–72, Toronto, Ontario, Canada, June 2010. Association for Computing Machinery.

[93] Chen Tian, Min Feng, and Rajiv Gupta. 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, pages 62–73, Toronto, Ontario, Canada, June 2010. Association for Computing Machinery.

[94] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or Discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 330–341, USA, November 2008. IEEE Computer Society.

[95] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. Association for Computing Machinery.

[96] Peng Tu and David A. Padua. Automatic Array Privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, Berlin, Heidelberg, August 1993. Springer-Verlag.

[97] Abhishek Udupa, Kaushik Rajan, and William Thies. Alter: Exploiting breakable dependences for parallelization. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 480–491, New York, NY, USA, 2011. ACM.

[98] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative Decoupled Software Pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, USA, September 2007. IEEE Computer Society.

[99] John Whaley and Monica 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 '04, pages 131–144, Washington DC, USA, June 2004. Association for Computing Machinery.

[100] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The SUIF Compiler System: a Parallelizing and Optimizing Research Compiler. Technical Report, Stanford University, Stanford, CA, USA, 1994.

[101] Michael E Wolf, Dror E Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 274–286. IEEE, 1996.

[102] Victor A Ying, Mark C Jeffrey, and Daniel Sanchez. T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware. In *47th Annual International Symposium on Computer Architecture (ISCA)*, pages 159–172. IEEE, 2020.

[103] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 290–301, February 2008. ISSN: 2378-203X.