Per*spec*tive: A Sensible Approach to Speculative Automatic Parallelization

Sotiris Apostolakis
Princeton University

Ziyang Xu
Princeton University

Greg Chan
Princeton University

Simone Campanoni
Northwestern University

David I. August
Princeton University

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 non-speculative automatic parallelization. The introduction of speculative automatic parallelization overcame these applicability limitations, but, even in the case of no misspeculation, these speculative techniques exhibit high communication and bookkeeping costs for validation and commit. This paper presents Per*spec*tive, a speculative-DOALL parallelization framework that maintains the applicability of speculative techniques while approaching the efficiency of non-speculative ones. Unlike current approaches which subsequently apply speculative techniques to overcome the imprecision of memory analysis, Per*spec*tive combines a novel speculation-aware memory analyzer, new efficient speculative privatization methods, and a planning phase to select a minimal-cost set of parallelization-enabling transforms. By reducing speculative parallelization overheads in ways not possible with prior parallelization systems, Per*spec*tive obtains higher overall program speedup (23.0× for 12 general-purpose C/C++ programs running on a 28-core shared-memory commodity machine) than Privateer (11.5×), the prior automatic DOALL parallelization system with the highest applicability.

CCS Concepts • **Software and its engineering → Compilers**; **Multithreading**.

Keywords automatic parallelization; speculation; privatization; memory analysis

1 Introduction

Using PThreads [42], Map-Reduce [9], OpenMP [31], and other libraries and languages, programmers routinely produce coarse-grained parallel (CGP) programs even at the warehouse scale. 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 developments in parallel programming languages, parallel libraries, and parallelizing compilers, reliably finding parallelism appropriate for multicore remains a challenge. Programs with CGP are not ideally suited for multicore as they tend to stress multicore's shared resources, such as caches and memory bandwidth. 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 transforms 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 transforms use control flow and data flow facts from analysis to make the code amenable to a given parallelization pattern. Examples of enabling transforms 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 among the most desirable is DOALL, the independent execution of loop iterations.

For decades, parallelizing compilers only performed enabling transforms that could be proven correct with respect to the facts provided by static analyses [2, 4, 7, 32, 37, 47]. While this approach showed some success in scientific codes, its reliance on memory analysis, a type of analysis notorious for its imprecision [17, 25], 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 [21, 23, 27, 36, 43]. 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 flow speculation is a popular speculative enabling transform that asserts the absence of flow dependences not manifested (or manifested infrequently) during profiling, backing its assertions with runtime checks to initiate misspeculation recovery when necessary. Another important speculative enabling transform is speculative privatization proposed in Privateer [21]. 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 for automatic parallelization represents, challenges remain that prevent its widespread adoption [5, 14, 22, 33]. While memory flow speculation is popular, its relaxed program dependence structure comes with a high cost. Even without any misspeculation, validation of memory flow speculation requires instrumenting memory operations on every iteration to log or communicate speculative accesses to additional validation code. For larger 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 [21, 23, 39].

The goal of this work is to produce a parallelizing compiler that achieves the coverage of speculative parallelization techniques while also reducing its costs. The proposed system, called Per*spec*tive, is an automatic DOALL parallelization framework integrating a speculation-aware memory analyzer, efficient variants of speculative privatization, and a planning phase to select the cheapest set of DOALL parallelization enabling transforms. This approach avoids certain unnecessary speculation overheads of prior work.

In Per*spec*tive, the speculation-aware memory analyzer increases the impact of speculative assertions. In practice, Per*spec*tive often recognizes that an inexpensive speculative enabler addresses problems thought by prior systems to require additional and more expensive speculation. For example, by observing the pruning of a path never taken in profiling by control-flow speculation, the speculation-aware memory analyzer knows to remove memory dependences that exist as a result of the existence of the pruned path. In contrast, prior work systems are unaware of these impacts and unnecessarily employ additional speculative techniques to remove those memory dependences.

Per*spec*tive's planning phase makes the addition of new speculative enablers feasible by enabling the selection of 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.

The primary contributions of this paper are:

- A novel **speculation-aware** memory analyzer that better leverages speculation by allowing memory analysis algorithms to interpret speculative assertions as program facts;
- New efficient **speculative privatization transforms** that avoid the overheads of prior speculative privatization techniques;
- A **planning** phase that combines non-speculative and speculative techniques to select the most profitable set of parallelization-enabling transforms;
- A **fully automatic** speculative DOALL parallelization framework for **commodity hardware** that exhibits **scalable** speedups by minimizing the speculative parallelization overheads of prior work.

Per*spec*tive achieves scalable automatic parallelization on commodity shared-memory machines without any programmer hints. We evaluate Per*spec*tive on a set of 12 C/C++ benchmarks used in prior state-of-the-art automatic parallelization system papers [4, 21, 23]. 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 [21]. These results come from the effective usage of static properties of the code in conjunction with cheap speculative assertions, the careful selection of applied transforms, and a lightweight process-based runtime system. Per*spec*tive represents an important step towards fulfilling the promise of automatic parallelization.

2 Background and Motivation

Software-based automatic DOALL parallelization systems have been studied for a long time. Early works including Polaris [2], SUIF [47], PD [38], and Hybrid Analysis [40] use static or runtime analysis to parallelize programs and examine the applicability of enabling transforms such as privatization and reduction. However, the imprecision of static analysis and the difficulty of extracting low-cost runtime

```
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 1. Motivating example from dijkstra [16]

checks limit the applicability of these systems to scientific codes.

More recent works like STMlite [27], Cluster Spec-DOALL [23], Privateer [21] use profile-guided speculation to overcome the limitations of static analysis and enable parallelization of loops with pointers, irregular memory accesses, and complex control flows. Among these works, Privateer [21] supports speculative privatization and reduction even in the presence of unrestricted pointers by using speculative heap separation, and has greater applicability than other 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 [5, 14]. This paper first identifies core inefficiencies of the state-of-the-art parallelizing compilers and subsequently describes how Per*spec*tive mitigates them.

2.1 Overheads of State-of-the-Art

Privateer [21] is the most applicable automatic DOALL parallelization system, and thus this paper focuses on the parallelization overheads of Privateer. Examining the evaluation

of Privateer, we identified two significant overheads: *(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, especially speculative ones. In §2.1.1 and §2.1.2, we discuss 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 [16]. Simplified code for the hot loop of this benchmark is shown in Figure 1. Section 3.4 shows how Per*spec*tive avoids these overheads in this example.

2.1.1 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 [21]. 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 parallelization systems, similarly to Privateer, cannot avoid the use of memory speculation in this case. However, Per*spec*tive is able to remove this dependence without the use of expensive-to-validate memory speculation, as shown in §3.4.

2.1.2 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 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 in order to track in which iteration each byte of the object was last written. In contrast, Per*spec*tive avoids this unnecessary bookkeeping by selecting a more efficient privatization variant, as discussed in §3.4.

3 The Per*spec*tive Approach

Current parallelizing compiler designs utilize memory analysis and speculative techniques independently and apply a fixed sequence of transforms with a focus on parallelization applicability rather than profitability. These designs lead to unnecessary overheads that often negate the benefits of parallelization. To overcome these limitations, Per*spec*tive

introduces a planning phase that involves careful selection of applied transforms and tight coupling of static analysis with speculation techniques. Per*spec*tive's design enables the discovery of efficient parallelization opportunities not possible in prior parallelizing compilers.

3.1 Planning

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

Enabling transforms modify the code to remove parallelization inhibitors, which in the context of DOALL parallelization are cross-iteration dependences. All transforms are split into two parts to facilitate planning: the applicability guard that participates in the planning phase of the compilation, and the actual transform that is applied, if selected, in the transform phase. The applicability guard utilizes properties produced by memory analysis and speculative assertions[1] to determine which parallelization inhibitors the corresponding transform can handle. The interface between memory analysis and speculative assertions, and enabling transforms is an annotated program dependence graph (PDG). The output of each applicability guard is collected in a transform proposal that is sent to a transform selector. Each proposal also includes a cost for the application of the transform and a set of speculative assertions required for the transform to be applicable and correct.

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

At the end of the planning phase, the transform selector picks a minimal-cost set of transform proposals necessary for DOALL parallelization.

3.2 Speculation-Aware Memory Analyzer

Prior techniques use memory analysis and speculative assertions independently. Instead, this work proposes a speculation-aware memory analyzer that combines the strengths of static analysis and cheap-to-validate speculative assertions to reduce the need for expensive speculation. If memory analysis fails on its own to resolve an analysis query, it interprets cheap-to-validate speculative assertions as facts, ignoring the possibility of misspeculation. The use of speculation necessitates the declaration of any speculation assertions leveraged in the process, for each answer.

Note that querying memory analysis after applying speculation would not have the same effect since the possibility of misspeculation restrains static analysis. For more details regarding speculation-aware memory analysis see §4.1.

3.3 New Enabling Transforms

Prior work on speculative parallelization focuses on the enabling effects of transforms without enough consideration of their costs. To maximize applicability, enabling transforms 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 transforms. By exposing a combination of static analysis information and the effect of speculative assertions, Per*spec*tive enables more efficient enabling transforms. This is especially true for the case of speculative privatization, efficient variants of which are explored in this paper. This section first discusses speculative privatization as it appears in prior work and then describes new variants that perform more efficient privatization.

Prior software speculative systems with extended support for privatization [21, 23] only infer the basic privatization property: a memory object does not have any cross-iteration data flows. 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 specify which worker last wrote each byte.

To avoid expensive monitoring of write sets during parallel execution and minimize copy-out costs, we propose four efficient variants of this transform. These variants require additional memory object properties apart from the basic privatization property to be applicable. Private objects could additionally be (a) independent (no loop-carried false dependences); (b) overwritten (written to the same locations at every loop iteration); (c) predictable (predictable live-out content); or (d) local (allocated outside the loop, but all accesses are contained within the loop).

Inference of any of these four private properties allows 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. [45] but were limited to static analysis based detection of privatization. Unlike any prior work, Per*spec*tive extends the applicability of these two variants with the usage of speculative assertions to programs with pointers, dynamic allocation, and type casts.

3.4 Example

This section describes how the new ideas introduced in Per*spec*tive (§3.1, §3.2, §3.3) enable efficient parallelization of the dijkstra benchmark. This section also highlights the limitations of prior work. Consider the code again in
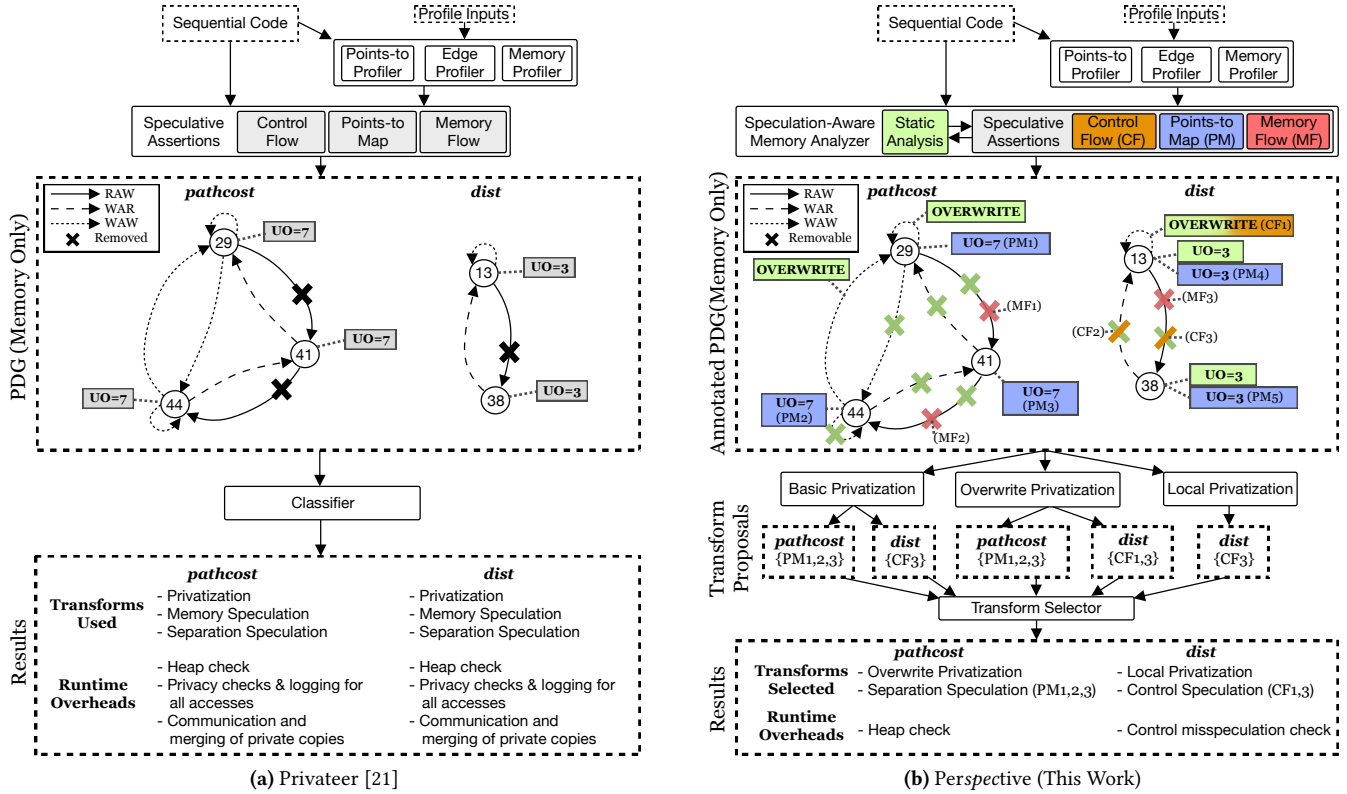
---

[1]Speculative assertions are predictions of program properties based on profile information.

(a) Privateer [21]

(b) Perspective (This Work)

Figure 2. Comparison of the decision process for handling memory objects `pathcost` and `dist` of dijkstra. Numbers in circles are line numbers in Figure 1; "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.

Figure 1, briefly discussed in §2.1. Figure 2 compares the compilation flow of Perspective with that of Privateer for handling memory objects `pathcost` and `dist`.

Perspective employs an exploration phase that yields a much more profitable plan than Privateer's scheme. At first, the **speculation-aware memory analyzer (SAMA)** processes the sequential code and profile information and produces a program dependence graph (PDG) annotated with properties and underlying speculative assertions, as shown in Figure 2b.

In this example (Figure 1), the branch in line **11** is heavily biased; it is always taken in practice. Therefore, control speculation leveraging edge profiling information can assert 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, our speculation-aware memory analyzer can view this assertion as a fact, ignore the branch, 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 memory speculation in any prior speculative parallelization system.

Apart from providing different options for removable edges, SAMA also provides useful information for non-removable edges. Non-removable output dependence edges, in the example in Figure 2b, are annotated with the *overwrite* property that indicates that the destination operation always overwrites the footprint of the source operation.

Furthermore, SAMA annotates nodes of instructions that may access memory with underlying memory objects, namely the instruction's memory footprint ("UOs" in Figure 2b). This information can be inferred either statically or via points-to profile information, and is required for privatization transforms 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 transforms offer to handle memory objects, including **overwrite privatization** and **local privatization**, proposed in §3.3.

```
10  int dequeue() {
11    if (!nullQHead()) {
12      // Privacy Local Check & Logging
13      private_write(&dist, sizeof(int), md); // < 1% added OH
14      dist = ...
15      ...
16    }
17
18
19  }
20
21  void worker_loop(int start, int N, int step) {
22    void *md = alloc(); // allocate metadata
23    // Separation Local Check
24    check_heap(pathcost, PRIVATE); // < 0.001% added OH
25    check_heap(&dist, PRIVATE); // < 0.001% added OH
26    for (src=start; src<N; src+=step) {
27      // Privacy Local Check & Logging
28      private_write(pathcost, N*sizeof(int), md); // < 1% added OH
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++) {
36          // Privacy Local Checks & Logging
37          private_read(&dist, sizeof(int), md); // 11.4% added OH
38          private_write(&nDist, sizeof(int), md); // 13.3% added OH
39          nDist = adj[v][i] + dist;
40          // Privacy Local Check & Logging
41          private_read(&pathcost[i], sizeof(int), md); // 11.6% added OH
42          if (pathcost[i] > nDist) {
43            // Privacy Local Check & Logging
44            private_write(&pathcost[i], sizeof(int), md); // < 1% added OH
45            pathcost[i] = nDist;
46            enqueue(i, nDist);
47          }
48        }
49      }
50      if (checkpointDue()) {
51        checkPrivAcccessesConflicts(md); // < 1% added OH
52      }
53    }
54    communicateLiveOutMemState(md); // < 1% added OH
55  }
```

```
10  int dequeue() {
11    if (!nullQHead()) {
12
13
14      dist = ...
15      ...
16    }
17    else
18      misspec("Control misspec in dequeue()"); // 0% added OH
19  }
20
21  void worker_loop(int start, int N, int step) {
22
23    // Separation Local Check
24    check_heap(pathcost, OVERWRITE_PRIVATE); // < 0.001% added OH
25
26    for (src=start; src<N; src+=step) {
27
28
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++) {
36
37
38
39          nDist = adj[v][i] + dist;
40
41
42          if (pathcost[i] > nDist) {
43
44
45            pathcost[i] = nDist;
46            enqueue(i, nDist);
47          }
48        }
49      }
50    }
51    // only last iter's pathcost array
52    // needs to be communicated
53    if (src == N-1+step)
54      communicate_pathcost(); // < 1% added OH
55  }
```

(a) Privateer [21]   **(b)** Perspective (This Work)

Figure 3. Comparison of parallelized code for dijkstra. Logging and checks during loop execution dominate the overheads, indicated in the code as "added OH".

The **DOALL planner** 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 transform (no monitoring and no copy-out costs), and its 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 transform.

On the other hand, Privateer, the prior automatic DOALL parallelization system with the highest applicability, is unable to parallelize *dijkstra* as efficiently as Per*spec*tive (Figure 2a). 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 remaining edges. This overall lack of information impedes consideration of the efficient privatization variants described in this paper. 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 3 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 the useful work of each worker. It is clear that Per*spec*tive is able to parallelize dijkstra with minimum overheads thanks to the use of speculative-aware memory analyzer, careful selection of applied transforms, and use of efficient privatization variants. In fact, Per*spec*tive exhibits 4.8× speedup over Privateer for dijkstra (see §5).

4 Framework Design and Implementation

Per*spec*tive is a framework for DOALL parallelization that incorporates the ideas described in §3. Figure 4 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
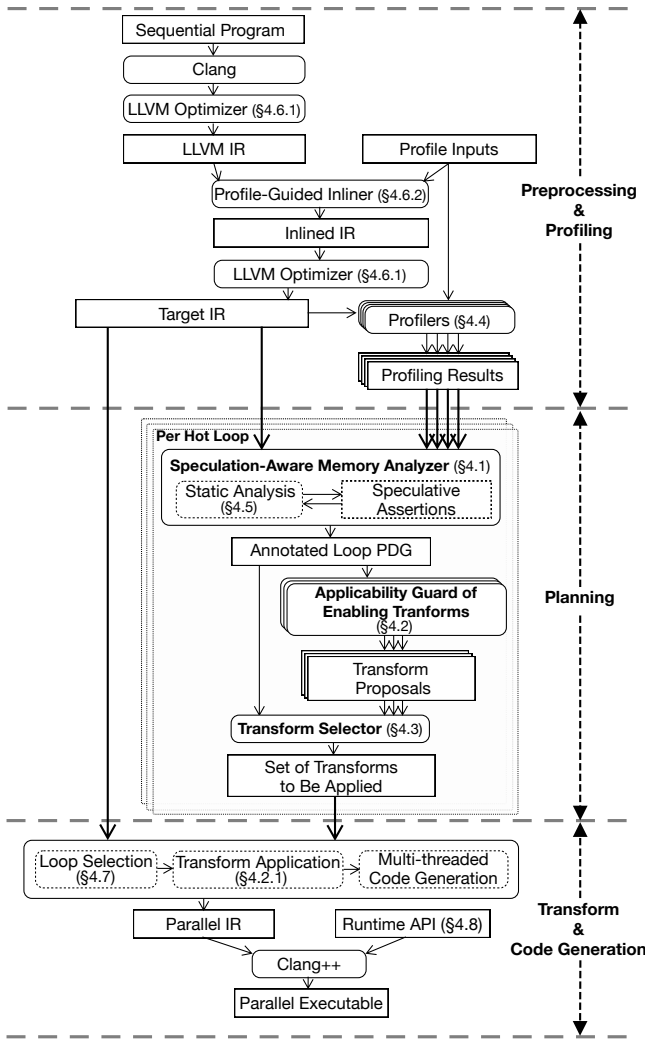
Figure 4. Perspective Framework Overview

which the code is canonicalized, and the profiling results are generated. In the planning phase, for each hot loop, the speculation-aware memory analyzer is queried to annotate the PDG with properties regarding dependences and instructions. The applicability guard of each enabling transform examines the annotated PDG and creates transform proposals that offer to remove parallelization-inhibiting cross-iteration dependences in the loop, along with their cost. Then, the transform selector considers these proposals and selects a minimal-cost set of enabling transforms, if a profitable DOALL plan is available. Finally, the compiler selects a set of compatible parallelizable loops with the maximum profitability, applies the transforms 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 section begins with the description of components in the planning phase, which includes the main contributions of this paper, and subsequently describes other parts of the framework.

4.1 Speculation-Aware Memory Analyzer

The speculation-aware memory analyzer is queried 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. The memory analyzer does not make any decisions but instead presents all alternatives for each inferrable property.

Making memory analysis aware of cheap-to-validate speculation reduces the need for expensive memory speculation.

Perspective's memory analysis (CAF [20]) is composed of simple analysis algorithms that collaboratively resolve queries. The modularity of the analysis simplifies the addition of speculation awareness. Only analysis algorithms that could benefit from the collaboration need to be extended with a speculative mode that interprets speculative assertions.

Perspective includes analysis passes extended with awareness of speculative assertions from two profilers: edge profiler (detects biased branches) and value-prediction profiler. The rest of the speculative assertions are used independently of static analysis. While the use of these two types of speculative assertions in conjunction with static analysis already enables up to 2.5× improvement over a system with no collaboration of memory analysis and speculation (see §5), exposing even more cheap-to-validate speculative assertions to memory analysis could also be beneficial. Such exploration is left for future work.

Perspective uses edge profiling to produce a speculative control flow. Two examples of analysis passes that benefit from this speculative information is the kill-flow and the unique access paths (UAP) algorithms. Kill-flow analysis disproves memory dependences by finding killing operations along all feasible paths between two operations. If kill-flow interprets speculative control flow information, the feasible paths may be reduced, and kill-flow can assert the absence of a statically non-disprovable memory dependence. UAP collects a points-to set of objects for a pointer stored to a non-captured memory location (i.e., address never stored into memory and never passed to an externally defined function). The use of speculative control flow information enables the detection of speculatively dead stores in this set, decreasing its size and thus simplifying alias queries for this pointer.

Perspective uses value-prediction profiling to predict the result of certain load operations. If memory analysis passes assume that these predictions are correct, then they can re-interpret a predicted load as a store of the predicted value. One analysis algorithm that can benefit from that is again kill-flow. Kill-flow treats the predictable load as a kill operation for must-aliasing data flows.

4.2 Enabling Transforms

Enabling transforms address memory, register or/and control cross-iteration dependences.

4.2.1 Memory Dependences

Applicability: The applicability guard of a transform uses the results of the speculation-aware memory analyzer to determine which memory objects satisfy the properties required by the transform and records the speculative assertions used.

Transform Proposal: The output of each applicability guard is assembled in a transform proposal that is sent to the transform selector (§4.3). The proposal includes for each memory object an estimated handling cost based on the transform itself and the validation cost of all used speculative assertions. For simplicity, each transform 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 transforms and speculative assertions in our framework and in the context of DOALL parallelization, this simplified cost model proved sufficient for the detection of minimal-cost plans.

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

Separating objects is essential for two reasons. First, each transform 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 transform's heap is sufficient to validate underlying object assertions. This idea of heap separation has been explored previously by Johnson et al. (Privateer [21]).

Memory-related enabling transforms include:

- Privatization: Applicable for objects with no false dependences. Requires costly logging and merging at commit.
- Reduction: Applicable for objects that only participate in reduction operations. 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.
- Read-only: Applicable for objects that are unmodified within the loop. Requires no transform-specific checks.
- I/O deferral: Applicable for shared I/O objects. 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 paper introduces four efficient variants of the privatization transform. Their applicability criteria are described in §3.3.

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

4.2.2 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 transform cost, all transforms 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.

4.3 Transform Selector

The selector picks the cheapest set of transforms that enables DOALL parallelization. Its inputs are an annotated loop-centric PDG (generated by the speculation-aware memory analyzer) and the transform proposals of the enabling transforms. Given these inputs, it greedily selects the cheapest transform proposal for each memory object and for each cross-iteration register and/or control dependence. Though one could always increase the complexity of this selection, we have not found empirical evidence that justifies such extra complexity. If there is a memory object, or a cross-iteration register or control dependence that cannot be addressed by any enabling transform, then the selector concludes that DOALL is not applicable. Note that if no speculative assertions are used, it produces non-speculative plans, forgoing the need for any speculation overhead.

4.4 Profiling

Per*spec*tive uses a set of profilers to generate speculative assertions: (i) an edge profiler [26] that identifies biased branches and produces a speculative control flow; (ii) a memory flow dependence profiler [6] that asserts the absence of non-observed data flows; (iii) a value-prediction profiler [15]

that detects predictable loads; (iv) a pointer-to-object profiler [21] that produces a points-to map for detection of underlying objects for every memory access; and, (v) an object lifetime profiler [21] that detects short-lived memory objects, namely objects that exist only within a single loop iteration.

4.5 Static Analysis

Per*spec*tive uses a state-of-the-art memory dependence analysis framework tailored for parallelization (CAF [20]), in which multiple simple analysis algorithms collaboratively attempt to disprove dependences and minimize the need for speculation. The utilized analysis algorithms reason about shape analysis, reachability, flow killing, induction variables, scalar evolution of pointers, and particular features of the LLVM IR and the C standard library. Next, this section presents an improvement to the Kill-Flow algorithm of CAF.

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 exponentially many paths, its search is restricted to blocks which 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 1. 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. We extend the Kill-Flow algorithm of prior work [20] to detect this pattern. Observe that the loop header of the inner loop in line **29** dominates the read in line **42**. The extended Kill-Flow 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 the memory addresses read in line **42** at every iteration. This extension allows us to disprove additional data flows compared to the state-of-the-art and to further reduce the need for memory speculation.

4.6 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 [26] 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.

4.6.1 LLVM optimizations

Transforms 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 transforms are meant for optimized sequential execution. Some of these optimizations 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 set of LLVM IR enabling transforms that simplify and canonicalize the IR.

4.6.2 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 DOALL parallelization.

4.7 Loop Selection

An execution time profiler, similar to gprof [41], 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 our current implementation).

4.8 Runtime

Per*spec*tive includes an efficient runtime for both speculative 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 copy-on-write (CoW) semantics of processes to achieve low overhead for communicating live-in values from the main process to the workers. It also gives an implicit separation between the speculative states of the workers and the committed state that the main process maintains when speculation is used. Benefits of process-based parallelization have also been discussed by prior work [12, 21, 36]. To facilitate

Benchmark	Suite	% of Execution Time (Theoretical Speedup) (A)	SAMA'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	Perspective	Privateer	v1	v2	Perspective
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 1. Benchmark Details: (A) % of program execution time spent inside parallelized loop(s). Theoretical speedup calculated using Amdahl's Law with 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 proposed speculative privatization transforms. (D) Monitored read and write set sizes for each benchmark; v1 represents Per*spec*tive without proposed enablers and SAMA (planning only); v2 represents Per*spec*tive without SAMA.

cheap heap assignment validation, each worker's virtual memory address space is segmented into disjoint sections, corresponding to each transform's heap, enabling cheap heap assignment checks, same as in Privateer [21]. To avoid the overhead of process spawning for parallelized inner loops with multiple invocations (052.alvinn), fork() is used only once at program startup and each worker's virtual memory remapped at the start of every invocation.

Use of Shared Memory: The runtime system utilizes POSIX named shared memory in /dev/shm to share data among workers and the main process. For non-speculative parallelization plans, the *independent* privatization's heap (§4.2.1) 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 [18, 36], 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 for detection of 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.

5 Evaluation

We evaluate Per*spec*tive 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 Per*spec*tive compiler is implemented on the LLVM Compiler Infrastructure (version 5.0.2) [26].

We evaluate Per*spec*tive against 12 C and C++ benchmarks (Table 1), covering all the parallelizable (exhibiting speedup) benchmarks from two state-of-the-art automatic speculative-DOALL parallelization papers (Privateer [21], Cluster Spec-DOALL [23]) as well as an additional benchmark (179.art) from HELIX [4], a non-speculative automatic parallelization system. We choose benchmarks that have been parallelized in prior work because the goal of this work is to boost the efficiency of automatic parallelization while maintaining the applicability of prior work.

We modify the benchmarks from Polybench and the dijkstra benchmark from MiBench to dynamically allocate previously statically allocated arrays and accept command line-defined array sizes in the same way as prior work [21, 23]. 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. Reported speedups are an average of 5 runs to minimize, although very small, the effect of variance in execution time between runs.
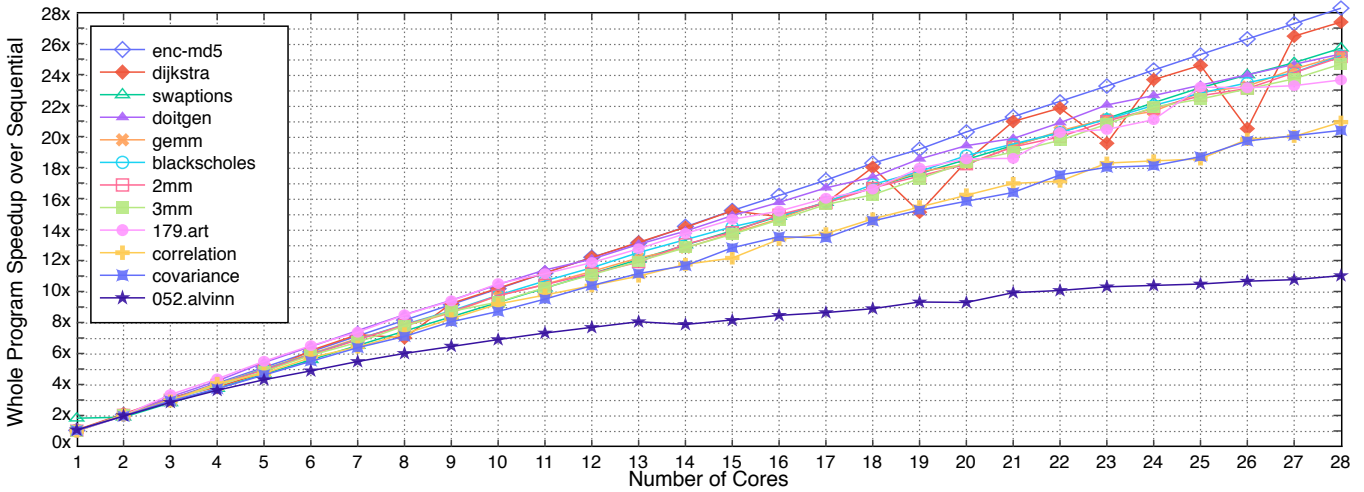
Figure 5. Per*spec*tive's Fully Automatic Whole Program Speedup over Sequential Execution

5.1 Scalability of Per*spec*tive

Figure 5 presents fully automatic whole program speedups across a various number of cores (up to 28 cores) for the 12 evaluated C/C++ benchmarks on our 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 on all the benchmarks thanks to the elimination of unnecessary overheads with the careful selection of applied transforms, the use of the speculation-aware memory analyzer, and the introduction of new enabling transforms.

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 transform 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, we observe speedups that exceed their theoretical limits, which we attribute to two factors: (1) The compiler replaces all calls to `malloc()` inside a parallelized loop with our own heap allocator. Our implementation of this allocator 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 [19]. This effect can be seen in the performance of `179.art`, `enc-md5`, and `doitgen`.

5.2 Comparison with State-of-the-Art

We compare Perspective with Privateer [21], the most applicable prior automatic speculative-DOALL system. We utilize the Privateer implementation provided by the authors of the Privateer paper with modifications to adhere to the more recent LLVM version used for Per*spec*tive. We achieve comparable speedups and runtime overheads compared to 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 §2.1. This is apparent in columns (D) of Table 1, 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 6 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 doubles the geomean speedup achieved by Privateer by minimizing unnecessary memory access monitoring and checks.

5.3 Performance Analysis of Per*spec*tive

We create two variants of Per*spec*tive with some components disabled to quantify the impact of the three main contributions of this paper (planning, new enablers, SAMA). The first variant (*Planning Only / v1*) includes everything in Per*spec*tive except for SAMA and the new enabling transforms (i.e., efficient privatization variants in §4.2.1). Without SAMA, 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 SAMA). Figure 6 compares
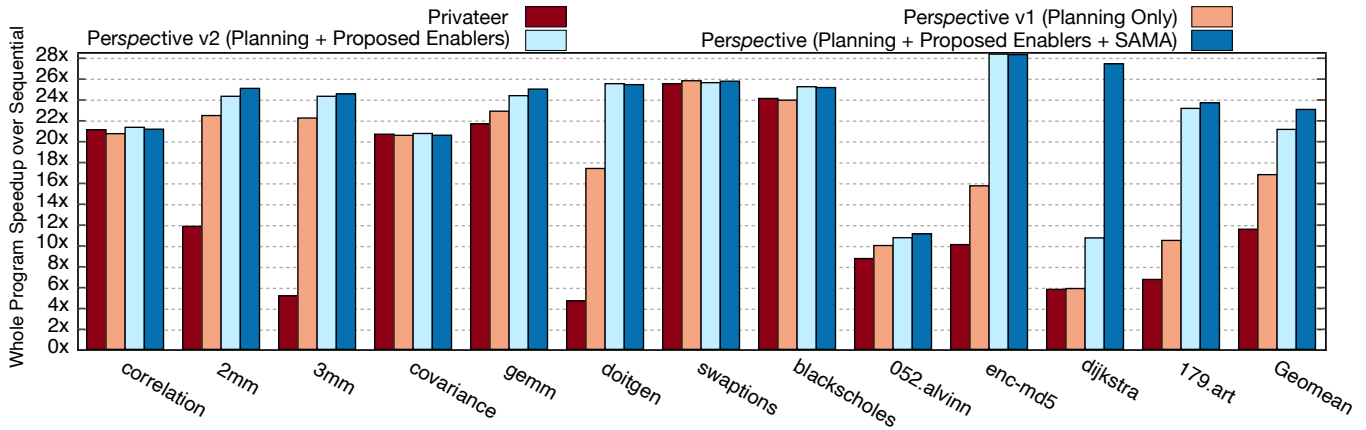
Figure 6. Whole Program Speedup Comparison among Variants of Per*spec*tive and Privateer with 28 Cores

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 transforms that need to be applied, as opposed to Privateer that overly aggressively applies speculative transforms. 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% additional geomean speedup compared to Privateer. The benefit of planning is particularly high for benchmarks with increased read set monitoring, including 2mm, 3mm, doitgen, enc-md5, and 179.art (Figure 6). 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 1 that for doitgen and 3mm, the monitored read set size is reduced from 2.53TB and 3TB, respectively, to zero. The absence of instrumentation of certain reads additionally enables a peephole optimization that hoists monitoring of writes to the same location outside a loop, further decreasing the overhead.

The introduction of new efficient privatization transforms (§3.3) 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 1. The performance impact for each benchmark depends on the amount of monitoring avoided, 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 the *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 as profitable. The planning phase is essential for exposing all the fine-grained information that makes these new transforms applicable and for allowing them to be selected over more expensive transforms.

The full version of Per*spec*tive additionally includes the speculation-aware memory analyzer (SAMA, §4.1). With SAMA, static analysis can remove by leveraging cheap-to-validate speculative assertions 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 §3.4). The introduction of SAMA also removes additional dependences from 179.art, enc-md5, and blackscholes, as shown in columns (B) of Table 1. 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 SAMA, but the performance impact is small.

5.4 Misspeculation Evaluation

Perspective uses only high confidence speculation to minimize the chance of misspeculation. Only properties that hold true without exception on the training inputs are speculated.

This conservative approach led to a complete lack of misspeculation on the evaluation inputs for the eight speculatively parallelized benchmarks (2mm, 3mm, doitgen, and 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
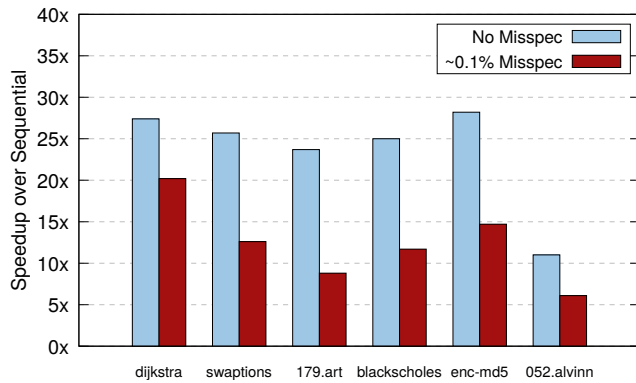
Figure 7. Impact of Misspeculation

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 [25].

Since none of the benchmarks exhibit misspeculation on the given inputs, we artificially inject misspeculation 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, we perform a weighted average of non-misspeculating and misspeculating runs to achieve an average corresponding to the desired rate.

Figure 7 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 supports the decision of using only high-confidence speculation in Per*spec*tive.

6   Related Work

Early non-speculative DOALL parallelizing compilers (Polaris[2], SUIF[1, 47]) are limited by imprecise static analysis. LRPD[39] and R-LRPD[8] overcome the limitations of static analysis by leveraging speculation. Yet, these works are restrained to array-based applications and cannot handle pointers and dynamic data structures.

More recent works (STMlite [27], CorD [43], Cluster Spec-DOALL [23], Privateer [21]) 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 [4], DOACROSS [7], DSWP [32], PS-DSWP [37]) that tolerate more dependences than DOALL parallelization by allowing communication among workers. Per*spec*tive targets only DOALL parallelism, but the contributions of this paper should be profitable to other parallelization paradigms as well. We leave the exploration of other parallelization schemes for future work.

Other works [10, 24, 30, 34, 35] propose systems that require developers to cast programs in specialized code patterns or insert annotations to express their intent better. Instead, Per*spec*tive fully automatically parallelizes general-purpose applications without the need for annotations or specialized abstractions.

Another line of work [3, 28, 44, 46] 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 [3, 28, 46] or by depending on user approval [44]. Instead, Per*spec*tive extracts parallelism without violating the sequential program semantics.

In terms of speculation-aware analysis, Devecsery et al. [11] and Fernández et al. [13] extend certain analysis algorithms with knowledge of profile-based speculative information. Yet, contrary to our speculation-aware memory analyzer, these works do not target automatic parallelization or even dependence analysis.

Neelakantam et al. [29] propose converting biased branches to assertions to allow subsequent transforms to leverage speculative control flow information. Instead, Per*spec*tive leverages speculative control flow information during analysis and planning, prior to transformation.

7   Conclusion

This work identifies and mitigates core inefficiencies of prior automatic speculative-DOALL systems. Per*spec*tive combines a novel speculation-aware memory analyzer, efficient variants of speculative privatization, and a planning phase to generate minimal-cost DOALL parallelization plans, avoiding overheads of prior work. Per*spec*tive fully-automatically yields geomean whole-program speedup of 23.0× over sequential execution for 12 C/C++ benchmarks on a 28-core shared-memory commodity machine, double the performance of Privateer, the prior automatic speculative-DOALL system with the highest applicability. Per*spec*tive represents an important advance in fulfilling the promise of automatic parallelization.

Acknowledgments

References

[1] Saman P. Amarasinghe and Monica S. Lam. 1993. 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)*. Association for Computing Machinery, Albuquerque, New Mexico, USA, 126–138. https://doi.org/10.1145/155090.155102

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

[3] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. HELIX-UP: Relaxing program semantics to unleash parallelization. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*. 235–245. https://doi.org/10.1109/CGO.2015.7054203

[4] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. 2012. HELIX: automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. Association for Computing Machinery, San Jose, California, 84–93. https://doi.org/10.1145/2259016.2259028

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

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

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

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

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

[10] Enrico A. Deiana, Vincent St-Amour, Peter A. Dinda, Nikos Hardavellas, and Simone Campanoni. 2018. Unconventional Parallelization of Nondeterministic Applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Association for Computing Machinery, Williamsburg, VA, USA, 432–447. https://doi.org/10.1145/3173162.3173181

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

[12] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. 2007. Software Behavior Oriented Parallelization. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 223–234. https://doi.org/10.1145/1250734.1250760

[13] Manel Fernández and Roger Espasa. 2002. Speculative Alias Analysis for Executable Code. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*. IEEE Computer Society, Washington, DC, USA, 222–231. http://dl.acm.org/citation.cfm?id=645989.674312

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

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

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

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

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

[19] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. 2011. 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)*. Association for Computing Machinery, Portland, Oregon, USA, 519–536. https://doi.org/10.1145/2048066.2048108

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

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

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

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

[24] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, San Diego, California, USA, 211–222. https://doi.org/10.1145/1250734.1250759

[25] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 4 (Dec. 1992),

323–337. https://doi.org/10.1145/161494.161501

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

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

[28] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2013. Parallelizing Sequential Programs with Statistical Accuracy Tests. *ACM Trans. Embed. Comput. Syst.* 12, 2s, Article Article 88 (May 2013), 26 pages. https://doi.org/10.1145/2465787.2465790

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

[30] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. 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)*. Association for Computing Machinery, Salt Lake City, Utah, USA, 499–512. https://doi.org/10.1145/2541940.2541964

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

[32] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*. IEEE Computer Society, Barcelona, Spain, 105–118. https://doi.org/10.1109/MICRO.2005.13

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

[34] Prakash Prabhu, Stephen R. Beard, Sotiris Apostolakis, Ayal Zaks, and David I. August. 2018. 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)*. Association for Computing Machinery, New York, NY, USA, Article Article 15, 12 pages. https://doi.org/10.1145/3243176.3243193

[35] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. 2011. 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)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/1993498.1993500

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

[37] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization (CGO '08)*. Association for Computing Machinery, Boston, MA, USA, 114–123. https://doi.org/10.1145/1356058.1356074

[38] Lawrence Rauchwerger and David Padua. 1994. 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)*. Association for Computing Machinery, Manchester, England, 33–43. https://doi.org/10.1145/181181.181254

[39] Lawrence Rauchwerger and David A. Padua. 1999. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (Feb. 1999), 160–180. https://doi.org/10.1109/71.752782

[40] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2003. Hybrid analysis: static & dynamic memory reference analysis. *International Journal of Parallel Programming* 31, 4 (Aug. 2003), 251–283. https://doi.org/10.1023/A:1024597010150

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

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

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

[44] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. 2009. 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)*. Association for Computing Machinery, New York, NY, USA, 177–187. https://doi.org/10.1145/1542476.1542496

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

[46] Abhishek Udupa, Kaushik Rajan, and William Thies. 2011. ALTER: Exploiting Breakable Dependences for Parallelization. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 480–491. https://doi.org/10.1145/1993498.1993555

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

A Artifact Appendix

A.1 Abstract

The artifact for this paper contains code and data to generate, with minimal effort, the main evaluation results of this paper and corroborate its claims. A Dockerfile is provided to create a docker image that can run across different platforms with all the software dependencies automatically installed. The artifact additionally includes a precompiled copy of the proposed compiler, along with variants and a state-of-the-art compiler (from prior work) used for comparison in our evaluation. Further, the artifact includes binaries for the parallel and sequential versions of the evaluated benchmarks, scripts to run them and reproduce speedup graphs (Figures 5 and 6 in the evaluation), along with the source code and a build system to regenerate these binaries. We also provide Collective Knowledge (CK) integration to simplify the interface of our experimentation workflow.

A.2 Artifact Check-list (meta-information)

- **Algorithm:** Speculative Automatic DOALL Parallelization
- **Program:** Includes benchmarks from PolyBench, PARSEC, MiBench, and Trimaran suites. SPEC CPU benchmarks are omitted to respect license restrictions.
- **Compilation:** Compilation of benchmarks is performed within the docker container. Required compilers are either automatically installed within the docker container or are included precompiled.
- **Transformations:** Automatic parallelization of sequential source code.
- **Binary:** Binaries for sequential and parallel versions of the benchmarks included (to execute within the docker container). Source code and scripts also included to regenerate binaries.
- **Data set:** Small inputs for the benchmarks are provided to reduce execution time and hardware requirements of the artifact evaluation.
- **Run-time environment:** Requires docker daemon running.
- **Hardware:** CPU-based x86_64 shared-memory machine with ≥ 4 physical cores, memory ≥ 4GB
- **Run-time state:** We recommend to disable TurboBoost or any other processor frequency related optimization for more stable speedup results. Ideally, minimal concurrent applications should run during experiments.
- **Execution:** Performed within a docker container and automated via CK command line.
- **Metrics:** Speedup of parallel execution over sequential execution, and comparison with prior work.
- **Output:** The output is two plots similar to Figures 5 and 6 from the evaluation section of this paper and a file with the speedup results in a text form (along with expected results as produced on our reference machine).
- **Experiments:** i) Scalability experiment for the proposed parallelizing compiler (evaluate with different number of cores (Figure 5); and, ii) comparison with state-of-the-art prior work and variants of the proposed compiler (Figure 6).
- **How much disk space required (approximately)?:** 10GB.

- **How much time is needed to prepare workflow (approximately)?:** Preparation takes only a few minutes (for building the docker image) if provided precompiled binaries for the evaluated benchmarks are used (default option). Preparation can take around 2 hours if profiling and sequential and parallel binaries are reproduced.
- **How much time is needed to complete experiments (approximately)?:** Experiments take around 3 hours in total (roughly 1.5 hours each one).
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** See LICENCE included in the archive file from https://doi.org/10.5281/zenodo.3606885.
- **Data licenses (if publicly available)?:** Varies for each benchmark suite. License notice can be found on the source code folder of each benchmark.
- **Workflow framework used?:** Collective Knowledge (CK), GNU make, and python3 scripts
- **Archived (provide DOI)?:** Yes, https://doi.org/10.5281/zenodo.3606885

A.3 Description

A.3.1 How Delivered

Download artifact from https://doi.org/10.5281/zenodo.3606885.

A.3.2 Hardware Dependencies

CPU-based x86_64 shared-memory machine with ≥ 4 physical cores, memory ≥ 4GB.

A.3.3 Software Dependencies

Installation of docker is required. All software dependencies are specified in the Dockerfile and are automatically installed in the docker image. Tested on macOS and Linux.

A.3.4 Data Sets

The inputs utilized in this artifact are smaller compared to the inputs used in the evaluation of this paper (§5). Inputs for producing Figures 5 and 6 are large enough for the sequential version to run for at least 10 minutes (as mentioned in §5). On the other hand, the evaluation inputs in the artifact lead to sequential times ranging from a few seconds up to 3 minutes. The goal for the artifact evaluation is to produce reasonable results while minimizing the required hardware resources (core count and memory size requirements) and the evaluation time.

A.4 Installation

A.4.1 Installation of Docker

We recommend the installation of Docker Community Edition (CE). Instructions for various platforms can be found here: https://docs.docker.com/install/. This section provides instructions for platforms on which we tested our artifact.

Ubuntu:

I. $ sudo apt-get update

II. `$ sudo apt-get install -y docker-ce`

Running the docker command requires root privileges. It can be run by a user without root privileges by adding the username to the docker group:
`$ sudo usermod -aG docker <username>`

macOS: To install docker on a macOS, download Docker Desktop for Mac from https://www.docker.com/products/docker-desktop and follow instructions.

A.4.2 Build Docker Image

I. make sure that the docker daemon is running
II. download the artifact from https://doi.org/10.5281/zenodo.3606885
III. decompress the tar archive and change to the decompressed directory (`asplos20ae_perspective`)
IV. build docker image:
`$ docker build --force-rm -t`
`asplos20ae_perspective:1.0 -f ./Dockerfile ./`

A.5 Experiment Workflow

I. run the built docker image:
`$ docker run -it --shm-size=1.5gb --memory=2gb`
`--memory-swap=-1 asplos20ae_perspective:1.0`
II. perform the experiments with the default parameters:
`$ ck run artifact`
(the latter command executes within the docker container)

A.6 Evaluation and Expected Result

Running the experiments produces graphs similar to Figures 5 and 6 in the evaluation section of this paper (§5). A text file with all the speedup results is also produced. The produced files can be found in the following (within the docker container) paths:

- /home/asplos20ae/scalability-exp.pdf
- /home/asplos20ae/comparison-exp.pdf
- /home/asplos20ae/result.txt

To copy files from the container to the host system do (outside the docker container):
`$ docker cp <containerID>:/home/asplos20ae/<file>`
`/path/in/host/.`
To find the container ID do:
`$ docker ps`
The artifact includes reference results as produced using the provided artifact on the machine used for the evaluation of this paper (shared-memory machine with two 14-core Intel Xeon CPU E5-2697 v3 processors running at 2.60GHz with TurboBoost disabled and 768GB of memory). The result in textual form (`result.txt`) includes the reference result along with the newly produced results. We also provide plots with the reference results (`/home/asplos20ae/reference-scalability-exp.pdf`, `/home/asplos20ae/reference-comparison-exp.pdf`).

For the comparison experiment, variant *v1* refers to *Perspective* with planning-only, *v2* refers to *Perspective* with planning and proposed enablers, and complete *Perspective* includes planning, proposed enablers, and SAMA (see §5).

Compared to the evaluation in this paper, this artifact utilizes smaller inputs, as discussed in §A.3.4. Smaller inputs lead to shorter execution times. This often translates to more variance, and lower speedups since one-time costs such as process spawning become more significant.

Note also that the reference results were produced in a machine with TurboBoost disabled, no concurrent applications running, and with the pinning of worker threads to separate cores. The pinning of worker threads to cores might not operate as expected in different platforms (e.g., two workers might be pinned to different virtual CPUs corresponding to the same physical CPU due to some unexpected core layout). Different hardware characteristics (e.g., disk, memory, CPU performance) certainly affect the achieved speedups.

Despite variances in the achieved speedups, across all platforms we tested, results demonstrate that the proposed work (*Perspective*) scales and yields significantly increased speedup over a state-of-the-art prior work (*Privateer*).

A.7 Experiment Customization

Experiments in this artifact are customizable. `ck` options allow for running tests with different configurations. Check the README and/or use `ck run artifact --help` to examine all the available options. Below is a partial list of configurations and their default values:

- **multicore:** (default: $1, 2, 3, 4$) Number of worker processes used in the scalability experiment.
- **compare:** (default: 4) Number of worker processes used in the comparison with prior work and variants experiment.
- **test_times:** (default: 2) Number of times to run each test (to reduce variance). The average execution time among these runs is reported.
- **from_X:** (default: `from_binary`) From what stage of the compilation to start the experiment (either use precompiled binaries if available, or perform code generation without reproducing profiles, or reproduce everything).

Some of these options may be turned off by prepending them with "no" (e.g. `--no-compare` will exclude running the comparison portion of the experiment).

A.8 Notes

Multiple README files are included in the artifact for more information. The README in the top-level directory contains the file/folder structure of the artifact with pointers to other README files within subdirectories.

A.9 Methodology

The artifact of this paper was reviewed according to the following guidelines: http://cTuning.org/ae/reviewing-20190109.html, https://www.acm.org/publications/policies/artifact-review-badging.