Perspective: A Sensible Approach to Speculative Automatic Parallelization

Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni†, and David I. August

ASPLOS 2020

PRINCETON UNIVERSITY

Northwestern University
Why Automatic Parallelization?
Why Automatic Parallelization?

Multicore systems are grossly underutilized [1,2]

Why Automatic Parallelization?

Multicore systems are grossly underutilized [1,2]

Extraction of parallelism fine-grained enough for multicore is notoriously hard [3]

Why Automatic Parallelization?

Multicore systems are grossly underutilized [1,2]

Extraction of parallelism fine-grained enough for multicore is notoriously hard [3]

Programmers are mostly limited to coarse-grained parallelism (CGP)

Why Automatic Parallelization?

Multicore systems are grossly underutilized [1,2]

Extraction of parallelism fine-grained enough for multicore is notoriously hard [3]

Programmers are mostly limited to coarse-grained parallelism (CGP)

CGP is ill-suited for multicore as it tends to stress multicore’s shared resources

The Potential of Automatic Parallelization: enable efficient use of multicore systems
The Potential of Automatic Parallelization: enable efficient use of multicore systems
Why Speculative Automatic Parallelization?
Why *Speculative* Automatic Parallelization?

For a long time, memory analysis limited applicability of automatic parallelization.
Why **Speculative** Automatic Parallelization?

For a long time, memory analysis limited applicability of automatic parallelization

- **undecidable in theory** [Landi, LPLS’92]
  
  For any fixed analysis algorithm, there is a counter-example input for which the algorithm is imprecise.
Why Speculative Automatic Parallelization?

For a long time, memory analysis limited applicability of automatic parallelization

- undecidable in theory [Landi, LPLS’92]
  For any fixed analysis algorithm, there is a counter-example input for which the algorithm is imprecise.

- insufficiently precise in practice [Hind, PASTE’01]
  especially for languages like C/C++.
Why **Speculative** Automatic Parallelization?

For a long time, memory analysis limited applicability of automatic parallelization

- **undecidable in theory** [Landi, LPLS’92]
  For any fixed analysis algorithm, there is a counter-example input for which the algorithm is imprecise.

- **insufficiently precise in practice** [Hind, PASTE’01]
  especially for languages like C/C++.

- **conservatively respects all possible inputs**
  Many real dependences rarely occur in practice.
Why Speculative Automatic Parallelization?

For a long time, memory analysis limited applicability of automatic parallelization

- undecidable in theory [Landi, LPLS’92]
  For any fixed analysis algorithm, there is a counter-example input for which the algorithm is imprecise.

- insufficiently precise in practice [Hind, PASTE’01]
  especially for languages like C/C++.

- conservatively respects all possible inputs
  Many real dependences rarely occur in practice.

Speculation overcame applicability limitations by enabling optimization of the expected case
Outline

Why Speculative Automatic Parallelization?

State-of-the-art Approach

Inefficiencies of State-of-the-art

The *Perspective* Approach

Evaluation

Conclusion
How to automatically parallelize?

State-of-the-art approach
How to automatically parallelize?

State-of-the-art approach

Sequential Source Code \(\rightarrow\) Static Analysis

Memory Analysis\(^1\)

\(^1\) Johnson et al., CGO '17
How to automatically parallelize?

State-of-the-art approach

Sequential Source Code → Static Analysis

Sequence of Enabling Transforms

1. Enabler 1
2. ... →
3. Enabler n

Memory Analysis

- (Speculative) Privatization
- (Speculative) Reduction
- Memory Speculation
- Control Speculation
- Value Prediction

Enabler 1

1. Johnson et al., CGO '17
2. Tu et al., LCPC '93
3. Johnson et al., PLDI '12
4. Mehrara et al., PLDI '09
5. Tian et al., PLDI '10
6. Kim et al., CGO '12
How to automatically parallelize?

State-of-the-art approach

Sequential Source Code → Static Analysis → Sequence of Enabling Transforms → Parallelization Transform → Parallelized code

- Memory Analysis
- (Speculative) Privatization
- (Speculative) Reduction
- Memory Speculation
- Control Speculation
- Value Prediction

Enabler 1 → ... → Enabler n

Memory Analysis

DOALL
PS-DSWP
HELIX

References:
1 Johnson et al., CGO '17
2 Tu et al., LCPC '93
3 Johnson et al., PLDI '12
4 Mehrara et al., PLDI '09
5 Tian et al., PLDI '10
6 Kim et al., CGO '12
7 Raman et al., CGO '08
8 Campanoni et al., CGO '12
How to automatically parallelize?

State-of-the-art approach

Sequential Source Code → Static Analysis → Sequence of Enabling Transforms → Parallelization Transform → Parallelized code

Disprove → Break → Tolerate

Sequence of Enabling Transforms:
- Enabler 1 → ... → Enabler n

(Speculative) Privatization
(Speculative) Reduction
Memory Speculation
Control Speculation
Value Prediction

DOALL
PS–DSWP
HELIX

It’s all about dependences

1. Johnson et al., CGO ’17
2. Tu et al., LCPC ’93
3. Johnson et al., PLDI ’12
4. Mehrara et al., PLDI ’09
5. Tian et al., PLDI ’10
6. Kim et al., CGO ’12
7. Raman et al., CGO ’08
8. Campanoni et al., CGO ’12
How to automatically parallelize?

State-of-the-art approach

Sequential Source Code \rightarrow \text{Static Analysis} \rightarrow \text{Parallelization Transform} \rightarrow \text{Parallelized code}

- Disprove
- Break
- Tolerate

Sequence of Enabling Transforms:

1. Enabler
2. \ldots
3. Enabler

(Speculative) Privatization
(Speculative) Reduction
Memory Speculation
Control Speculation
Value Prediction

Memories Analysis

- Johnson et al., CGO '17
- Tu et al., LCPC '93
- Johnson et al., PLDI '12
- Mehrara et al., PLDI '09
- Tian et al., PLDI '10
- Kim et al., CGO '12
- Raman et al., CGO '08
- Campanoni et al., CGO '12

It's all about dependences
How to automatically parallelize?

State-of-the-art approach

Sequential Source Code ➔ Static Analysis ➔ Parallelization Transform ➔ Parallelized Code

Disprove

Break

Tolerate

Sequence of Enabling Transforms

Enabler 1 ➔ ... ➔ Enabler n

(costs often negate parallelization benefits)

(Speculative) Privatization
(Speculative) Reduction
Memory Speculation
Control Speculation
Value Prediction

Memory Analysis

DOALL
PS–DSWP
HELIX

1 Johnson et al., CGO '17
2 Tu et al., LCPC '93
3 Johnson et al., PLDI '12
4 Mehrara et al., PLDI '09
5 Tian et al., PLDI '10
6 Kim et al., CGO '12
7 Raman et al., CGO '08
8 Campanoni et al., CGO '12
The most applicable prior automatic speculative DOALL system is Privateer*
The most applicable prior automatic speculative DOALL system is Privateer* with two identified inefficiencies.

* Nick P. Johnson et al., Speculative Separation for Privatization and Reductions in PLDI ‘12
The most applicable prior automatic speculative DOALL system is Privateer*.

Two identified inefficiencies:

Excessive use of memory speculation
Very expensive to validate due to costly communication and bookkeeping for each speculated dependence.

* Nick P. Johnson et al., Speculative Separation for Privatization and Reductions in PLDI ‘12
The most applicable prior automatic speculative DOALL system is Privateer*

two identified inefficiencies

Excessive use of memory speculation
Very expensive to validate due to costly communication and bookkeeping for each speculated dependence

Expensive speculative privatization
Monitor large write sets to correctly merge private memory states of parallel workers

* Nick P. Johnson et al., Speculative Separation for Privatization and Reductions in PLDI ‘12
Inefficiencies of state-of-the-art:

Overuse of expensive-to-validate memory speculation

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        i1:   *ptr = ...
    ...
    i2:   ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop
Inefficiencies of state-of-the-art:

Overuse of expensive-to-validate memory speculation

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        i1: *ptr = ...
    ...
    i2: ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

Program Dependence Graph (PDG)

- Cross-iter RAW Dep
- Cross-iter WAW Dep
- Intra-iter RAW Dep
Inefficiencies of state-of-the-art:

Overuse of **expensive-to-validate** memory speculation

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
    i1:  *ptr = ...
    ...
    i2:  ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

Program Dependence
Graph (PDG)

- **Cross-iter RAW Dep**
- **Cross-iter WAW Dep**
- **Intra-iter RAW Dep**

- **Iteration k**
  - `i1: *ptr = ...`
  - `i2: ... = ... + *ptr`

- **Iteration j**
  - `i1: *ptr = ...`
  - `i2: ... = ... + *ptr`
Inefficiencies of state-of-the-art:

Overuse of **expensive-to-validate** memory speculation  

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
    i1:  *ptr = ...
    ...
    i2:  ... = ... + *ptr
}
```

**Simplified example from the dijkstra benchmark (MiBench)**

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

**Program Dependence Graph (PDG)**

**DOALL parallelization applicability criterion:**
No cross-iteration dependences
Inefficiencies of state-of-the-art:

Overuse of \textit{expensive-to-validate} memory speculation

\begin{verbatim}
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        i1:  *ptr = ...
        ...
        i2:  ... = ... + *ptr
}
\end{verbatim}

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- \texttt{ptr} is not modified within the loop

Relaxing Program Dependence Graph (PDG)
Inefficiencies of state-of-the-art:

Overuse of **expensive-to-validate** memory speculation

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        i1:  *ptr = ...
    ...
        i2:  ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

Relaxing Program Dependence Graph (PDG)
Inefficiencies of state-of-the-art:

Overuse of **expensive-to-validate** memory speculation

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        *ptr = ...
    ...
    i2:  ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

Relaxing Program Dependence Graph (PDG)
Inefficiencies of state-of-the-art:

Overuse of expensive-to-validate memory speculation

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
    i1:   *ptr = ...
    ...
    i2:   ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

Relaxing Program Dependence Graph (PDG)
Inefficiencies of state-of-the-art:

Overuse of expensive-to-validate memory speculation

```c
for (i=0; i<N; ++i) {
...
  if (observed_always_true)
  i1:  *ptr = ...
  ...
  i2:  ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

Relaxing Program Dependence Graph (PDG)
Inefficiencies of state-of-the-art:

Overuse of \textit{expensive-to-validate} memory speculation

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        *ptr = ...
    ...
    i2: ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- \texttt{ptr} is not modified within the loop

Relaxing Program Dependence Graph (PDG)

- \textbf{Cross-iter RAW Dep}
- \textbf{Cross-iter WAW Dep}
- \textbf{Intra-iter RAW Dep}

DOALL-able but with use of \textit{expensive-to-validate} memory speculation
Inefficiencies of state-of-the-art:

Overuse of expensive-to-validate memory speculation

```
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true) {
        spec_write(ptr)
        i1:  *ptr = ...
    }
    ...
    spec_read(ptr)
    i2:  ... = ... + *ptr
}
```
Inefficiencies of state-of-the-art:

Overuse of expensive-to-validate memory speculation

```
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true) {
        spec_write(ptr)
        i1:    *ptr = ...
    }
    ...
    spec_read(ptr)
    i2:    ... = ... + *ptr
}
```
Inefficiencies of state-of-the-art:

Overuse of expensive-to-validate memory speculation

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true) {
        spec_write(ptr)
        i1:  *ptr = ...
    }
    ...
    spec_read(ptr)
    i2:  ... = ... + *ptr
}
```
Inefficiencies of state-of-the-art:

Overuse of **expensive-to-validate** memory speculation

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true) {
        spec_write(ptr)
        i1:  *ptr = ...
    }
    ...
    spec_read(ptr)
    i2:  ... = ... + *ptr
}
```

Worker 1  Worker 2  Validator
Iter 1     Iter 2
W,1,ptr    W,2,ptr  ok
Iter 3
R,3,ptr

Monitoring Overhead

Time

-- Iter 1 --

-- Iter 2 --

-- Iter 3 --
Inefficiencies of state-of-the-art:

Overuse of *expensive-to-validate* memory speculation

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true) {
        spec_write(ptr)
    }
    ...
    spec_read(ptr)
}
```

**Diagram:**

- **Worker 1**
  - **Iter 1**: W,1,ptr
  - **Iter 2**: W,2,ptr
  - **Iter 3**: R,3,ptr
- **Worker 2**
- **Validator**
  - ok
  - **Misspec**

*Monitoring Overhead*
Inefficiencies of state-of-the-art:

**Expensive speculative privatization**

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
    i1:  *ptr = ...
    ...
    i2:  ... = ... + *ptr
}
```

Assumptions
- branch condition **statically proven true**
- **ptr** is not modified within the loop
Inefficiencies of state-of-the-art:

Expensive speculative privatization

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        *ptr = ...
    ...
    i1:    *ptr = ...
    ...
    i2:    ... = ... + *ptr
}
```

Assumptions
- branch condition *statically proven* true
- `ptr` is not modified within the loop

Relaxing Program Dependence Graph (PDG)

- `Cross-iter RAW Dep`
- `Cross-iter WAW Dep`
- `Intra-iter RAW Dep`
Inefficiencies of state-of-the-art:

Expensive speculative privatization

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        *ptr = ...
    ...
    i2:  ... = ... + *ptr
}
```

Assumptions
- branch condition **statically proven** true
- `ptr` is not modified within the loop

Relaxing Program Dependence Graph (PDG)
Inefficiencies of state-of-the-art:

Expensive speculative privatization

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        i1:  *ptr = ...
    ...
    i2:  ... = ... + *ptr
}
```

Assumptions
- branch condition **statically proven** true
- `ptr` is not modified within the loop

Relaxing Program Dependence Graph (PDG)

- **Cross-iter RAW Dep**
- **Cross-iter WAW Dep**
- **Intra-iter RAW Dep**
Inefficiencies of state-of-the-art:

**Expensive speculative privatization**

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
    i1:  *ptr = ...
    ...
    i2:  ... = ... + *ptr
}
```

**Assumptions**
- branch condition **statically proven** true
- `ptr` is not modified within the loop

**Relaxing Program Dependence Graph (PDG)**
Inefficiencies of state-of-the-art:

Expensive speculative privatization

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        i1:  *ptr = ...
    ...
    i2:  ... = ... + *ptr
}
```

Assumptions

- branch condition **statically proven** true
- `ptr` is not modified within the loop

Relaxing Program Dependence Graph (PDG)

DOALL-able but expensive write monitoring used for live-out state
Inefficiencies of state-of-the-art:

**Expensive speculative privatization**

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true) {
        spec_write(ptr)
        *ptr = ...
    }
    ...
    i1: ... = ... + *ptr
}
```

Worker 1  Worker 2  Master

Time

Monitoring Overhead
Inefficiencies of state-of-the-art: Expensive speculative privatization

```
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true) {
        i1: spec_write(ptr)
            *ptr = ...
    }
    ...
    i2: ... = ... + *ptr
}
```
Inefficiencies of state-of-the-art:

Expensive speculative privatization

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true) {
        spec_write(ptr)
        *ptr = ...
    }
    ...
    i2: ... = ... + *ptr
}"
```
Inefficiencies of state-of-the-art:

**Expensive speculative privatization**

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true) {
        i1: spec_write(ptr)
            *ptr = ...
    }
    ...
    i2: ... = ... + *ptr
}
```

Monitoring Overhead

![Diagram showing execution flow and overhead](image)
Parallelization of dijkstra benchmark (MiBench) with Privateer*

Required monitoring of 973GB of reads & 649GB of writes for an input graph of 3K nodes! $O(N^3)$, where N is # of nodes

* Nick P. Johnson et al., Speculative Separation for Privatization and Reductions in PLDI ‘12
Outline

Why Speculative Automatic Parallelization?
State-of-the-art Approach
Inefficiencies of State-of-the-art
The Perspective Approach
Evaluation
Conclusion
Maintain the applicability of prior speculative automatic parallelization systems without unnecessary overheads
Fully leverage inexpensive speculative assertions to efficiently break dependences

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        *ptr = ... 
    ...
    i2:  ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

Program Dependence Graph (PDG)

- Green: Intra-iter RAW Dep
- Red: Cross-iter RAW Dep
- Green: Intra-iter WAW Dep
- Red: Cross-iter WAW Dep
Fully leverage inexpensive speculative assertions to efficiently break dependences

```c
for (i=0; i<N; ++i) {
  ...
  if (observed_always_true)
  *ptr = ... 
  ...
  ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

Program Dependence

Graph (PDG)

- **Cross-iter RAW Dep**
- **Cross-iter WAW Dep**
- **Intra-iter RAW Dep**
Fully leverage inexpensive speculative assertions to efficiently break dependences

```
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        *ptr = ...
    ...
    i2: ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

Program Dependence Graph (PDG)

![PDG Diagram]

- **Cross-iter RAW Dep**
- **Cross-iter WAW Dep**
- **Intra-iter RAW Dep**
Fully leverage inexpensive speculative assertions to efficiently break dependences

for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        *ptr = ...
    ...
    i2:    ... = ... + *ptr
}

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- \texttt{ptr} is not modified within the loop

Program Dependence Graph (PDG)
Fully leverage inexpensive speculative assertions to efficiently break dependences

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        *ptr = ...
    ...
    i2:  ... = ... + *ptr
}
```

Simplified example from the dijkstra benchmark (MiBench)

- branch condition cannot be statically proven true
- `ptr` is not modified within the loop

Program Dependence Graph (PDG)

![Program Dependence Graph](image)

DOALL-able without monitoring
Inexpensive control speculation check instead of monitoring

```c
for (i=0; i<N; ++i) {
    ...
    if (observed_always_true)
        *ptr = ...
    else
        misspec()
    ...
}
```
The Perspective Approach
The **Perspective** Approach

**Design goals**

- Increase awareness
- Enable collaboration
- **Avoid** unnecessary transforms
The **Perspective** Approach

- **Planning**
The **Perspective** Approach
The *Perspective* Approach

Diagram:
- Sequential Source Code → Static Analysis → Enabling Transforms
- Enabler 1 → ... → Enabler n
The Perspective Approach

Sequential Source Code → Static Analysis → PDG → Enabling Transforms → Transform Selector

Enabler 1
Enabler n

Planning
The *Perspective* Approach

- **Planning**

---

1. **Planning Phase**
2. **Transform Phase**
3. **Sequential Source Code** → **Static Analysis** → **Enabling Transforms** → **Transform Selector** → **Apply Transforms** → **Parallelized code**
The **Perspective** Approach

- Planning
- Speculation-Aware Memory Analysis

**Sequential Source Code** → **Static Analysis** → **Enabling Transforms** → **Transform Selector** → **Apply Transforms** → **Parallelized code**

- Planning Phase
- Transform Phase

Enabler 1 → ... → Enabler n
The **Perspective** Approach

- Planning
- Speculation-Aware Memory Analysis

#### Planning Phase
- Profile-based Speculative Assertions

#### Transform Phase
- Parallelized code
The **Perspective** Approach

- Planning
- Speculation-Aware Memory Analysis

**Sequential Source Code** → **Static Analysis** → **Enabling Transforms** → **Transform Selector** → **Apply Transforms** → **Parallelized code**

- Profile-based Speculative Assertions
- Planning Phase
- Transform Phase

- PDG
- Enabler 1
- Enabler n
The **Perspective** Approach

- **Planning**
- Speculation-Aware Memory Analysis

Diagram:
- Sequential Source Code → Static Analysis → Enabling Transforms
- Planning Phase
- Transform Selector → Apply Transforms → Parallelized code
- Profile-based Speculative Assertions
The Perspective Approach

- Planning
- Speculation-Aware Memory Analysis
- New Efficient Enabling Transforms

Sequential Source Code → Static Analysis → Enabling Transforms → Transform Selector → Apply Transforms → Parallelized code

Planning Phase
Transform Phase

Profile-based Speculative Assertions

Enabler 1

Enabler n

New Enablers

PDG
Revisiting motivating example with **Perspective**

Parallelization of dijkstra benchmark (MiBench)

- Excessive use of memory speculation
- Expensive privatization
- Required monitoring of 973GB of reads & 649GB of writes!

**4.8x speedup** over Privateer*

*Nick P. Johnson et al., PLDI ‘12
**Perspective Framework**
is implemented on the
LLVM Compiler Infrastructure

~80K loc in C/C++
Perspective’s Evaluation Methodology
Perspective’s Evaluation Methodology

Platform
Evaluated on a commodity shared-memory machine with 28 cores
Perspective’s Evaluation Methodology

Platform
Evaluated on a commodity shared-memory machine with 28 cores

Empirically Evaluated Claim
Maintain the applicability of prior automatic-DOALL systems while improving their efficiency
Perspective’s Evaluation Methodology

Platform
Evaluating on a commodity shared-memory machine with 28 cores

Empirically Evaluated Claim
Maintain the applicability of prior automatic-DOALL systems while improving their efficiency

Benchmarks
All parallelizable benchmarks from two state-of-the-art automatic DOALL-parallelization papers [1,2].
12 C/C++ benchmarks from SPEC CPU, PARSEC, PolyBench and MiBench.

Perspective yields scalable speedup
Perspective doubles performance of Privateer*

* Nick P. Johnson et al., Speculative Separation for Privatization and Reductions in PLDI ‘12
**Perspective** doubles performance of Privateer thanks to dramatic reduction of monitored reads/writes

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Monitored Read Set Size</th>
<th>Monitored Write Set Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Privateer</td>
<td>Perspective</td>
</tr>
<tr>
<td>enc-md5</td>
<td>1.87TB</td>
<td>39.1KB</td>
</tr>
<tr>
<td>052.alvinn</td>
<td>153GB</td>
<td>0B</td>
</tr>
<tr>
<td>179.art</td>
<td>1.6TB</td>
<td>0B</td>
</tr>
<tr>
<td>2mm</td>
<td>1TB</td>
<td>0B</td>
</tr>
<tr>
<td>3mm</td>
<td>3TB</td>
<td>0B</td>
</tr>
<tr>
<td>correlation</td>
<td>0B</td>
<td>0B</td>
</tr>
<tr>
<td>covariance</td>
<td>0B</td>
<td>0B</td>
</tr>
<tr>
<td>doitgen</td>
<td>2.53TB</td>
<td>0B</td>
</tr>
<tr>
<td>gemm</td>
<td>128MB</td>
<td>0B</td>
</tr>
<tr>
<td>blackscholes</td>
<td>0B</td>
<td>0B</td>
</tr>
<tr>
<td>swaptions</td>
<td>703KB</td>
<td>0B</td>
</tr>
<tr>
<td>dijkstra</td>
<td>973GB</td>
<td>0B</td>
</tr>
</tbody>
</table>
Perspective advances state-of-the-art by identifying and mitigating core inefficiencies of prior speculative automatic parallelization systems.

Perspective generates minimal-cost DOALL-parallelization plans by combining a planning phase, speculation-aware memory analysis, and efficient speculative privatization.

Perspective fully–automatically yields scalable speedup ($23.0 \times$ on 28 cores), double the performance of state-of-the-art.

Artifact available at: https://doi.org/10.5281/zenodo.3606885