Path Specialization: Reducing Phased Execution Overheads

Filip Pizlo, Erez Petrank, Bjarne Steensgaard
Purdue, Technion/Microsoft, Microsoft

ISMM'08 - Tucson, AZ

- Real-time, concurrent, and incremental garbage collectors are becoming main-stream techniques.

- But these collectors require *barriers* to be inserted, which causes execution to slow down.

- *Barriers slow down execution of programs.*
- This talk focuses on increasing the <u>throughput</u> of programs that use expensive barriers.

Types of Barriers

(a non-exclusive list of expensive barriers that we’re familiar with)

- Stopless (ISMM'07)
- Brooks read barrier (both lazy and eager)
- Yuasa barrier for concurrent or incremental mark-sweep

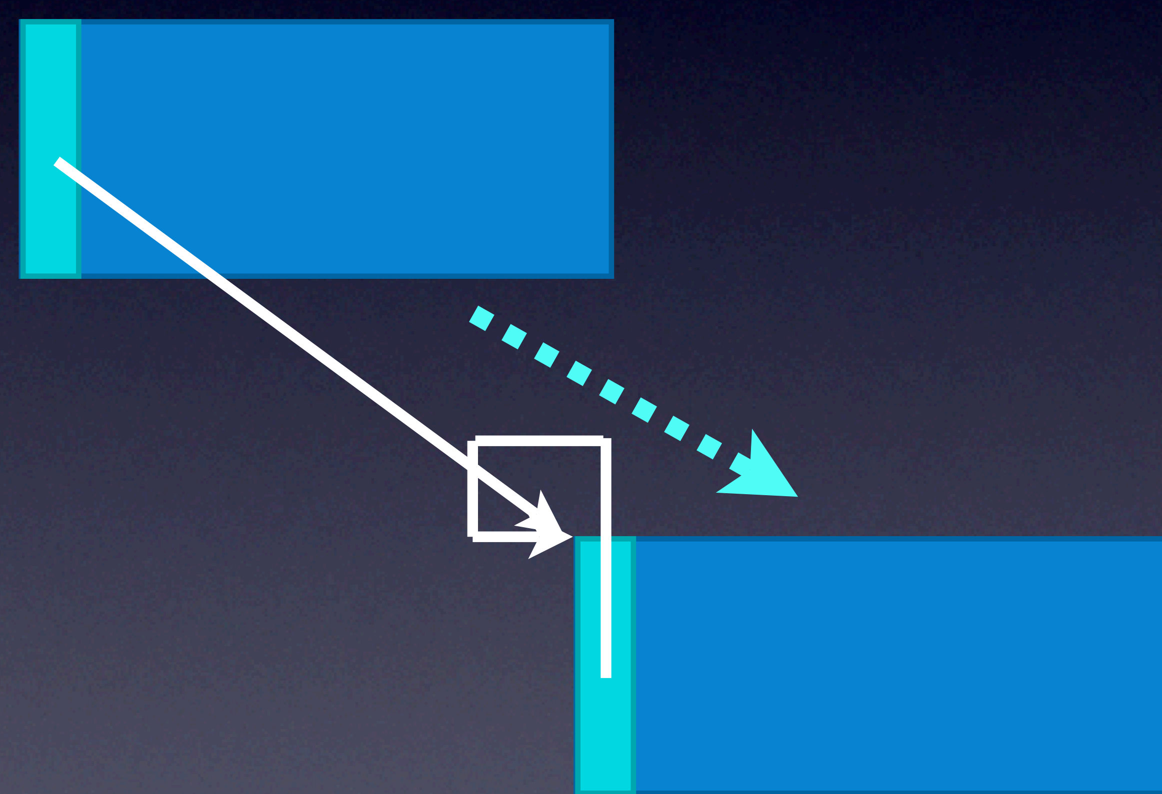
Stopless Barriers

- “The write barrier from heck” -anonymous
- Stopless barriers require potentially multiple branches, loads, stores, and CASes *even on primitive reads and writes*.
- But the barriers are only active during the (short) copying phase.

- Brooks read barriers
  - Useful when the mutator may see the same object in both to-space and from-space
  - Idea: each object has a pointer in its header to the “correct” version of the object.
  - This pointer may be self-pointing

Brooks Forwarding Pointer

Brooks Forwarding Pointer



“Lazy” Brooks

```
object a = b.f
use a
use a
```

```
object a = b.forward.f
use a.forward
use a.forward
```

These barriers are only needed when copying is ongoing.

Yuasa Write Barrier

a.f = b

```
if barrier active
   mark a.f
a.f = b
```

Yuasa Write Barrier

a.f = b

```
if barrier active
   mark a.f
a.f = b
```

We use this barrier in concurrent and incremental mark-sweep collectors.

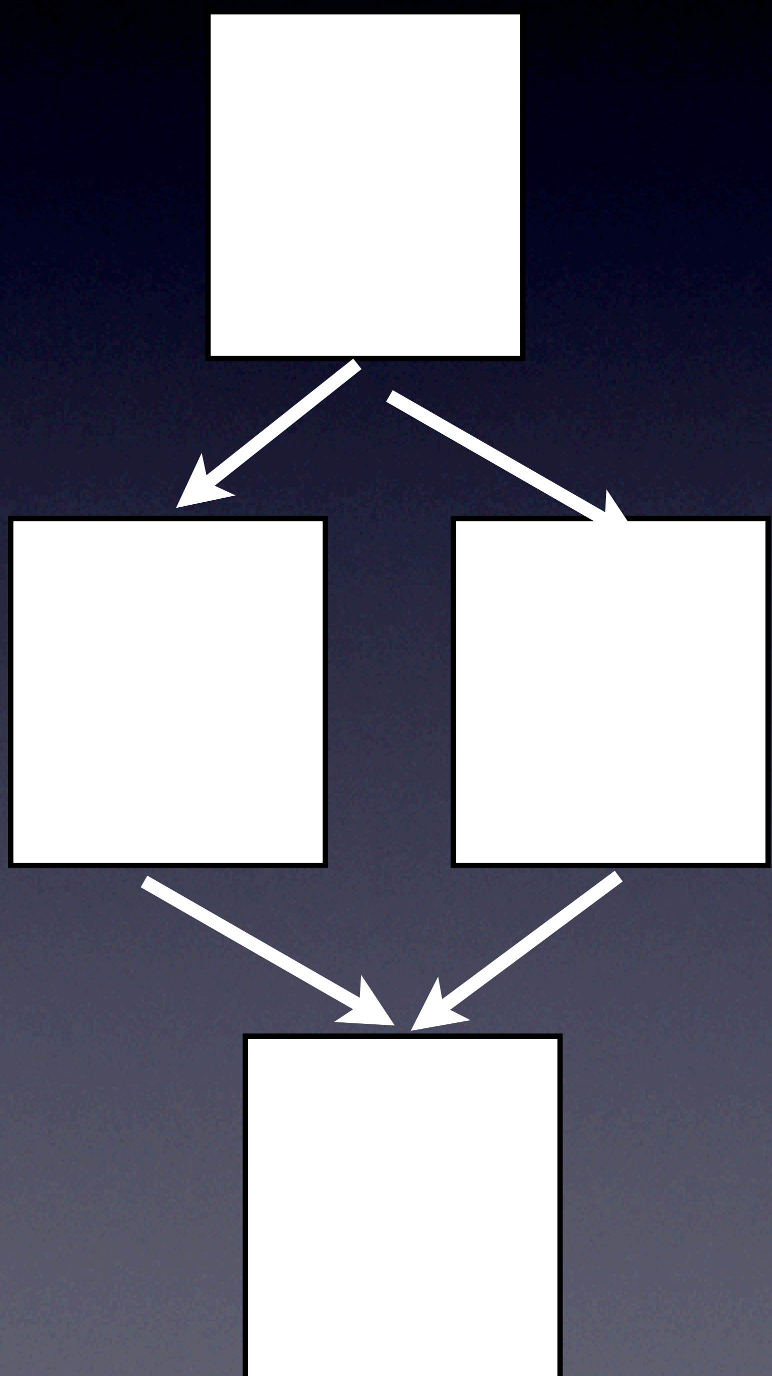
- Barriers for concurrent and incremental collectors tend to only be active during some phase of collector execution.
    - *Even if the collector is always running, the barriers are only active a fraction of the time.*
- Concurrent Mark-sweep: only active during *marking* phase.
- Metronome: Brooks only active during the (rare) copying phase
- Stopless: only active during the (rare and short) copying phase.

- What we want:
  - Make code run faster when the barriers are not needed.
  - Make code run not much slower when the barriers <u>are</u> needed.
  - Result: *get better throughput.*

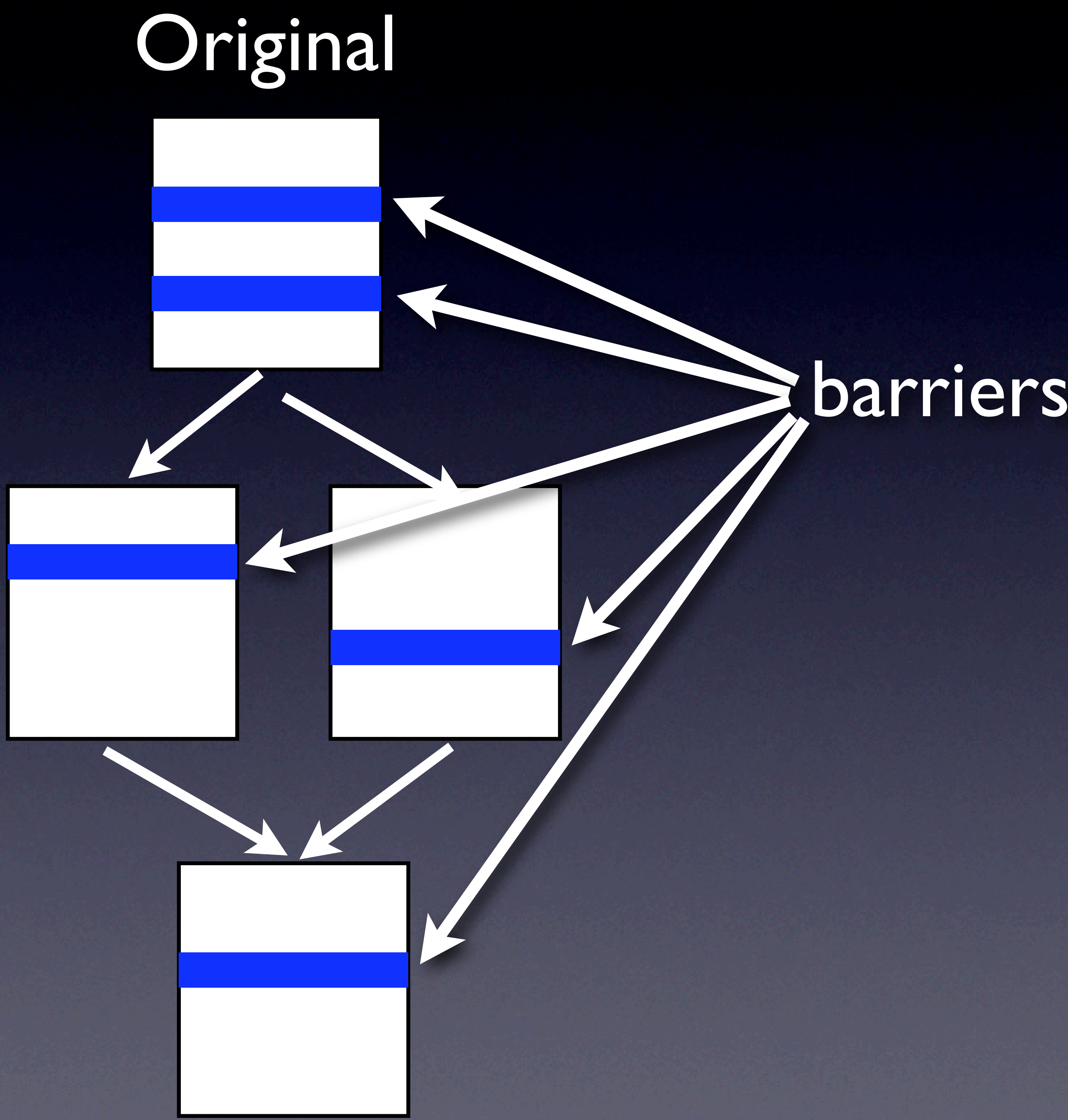
Path Specialization

Simple Example

Original

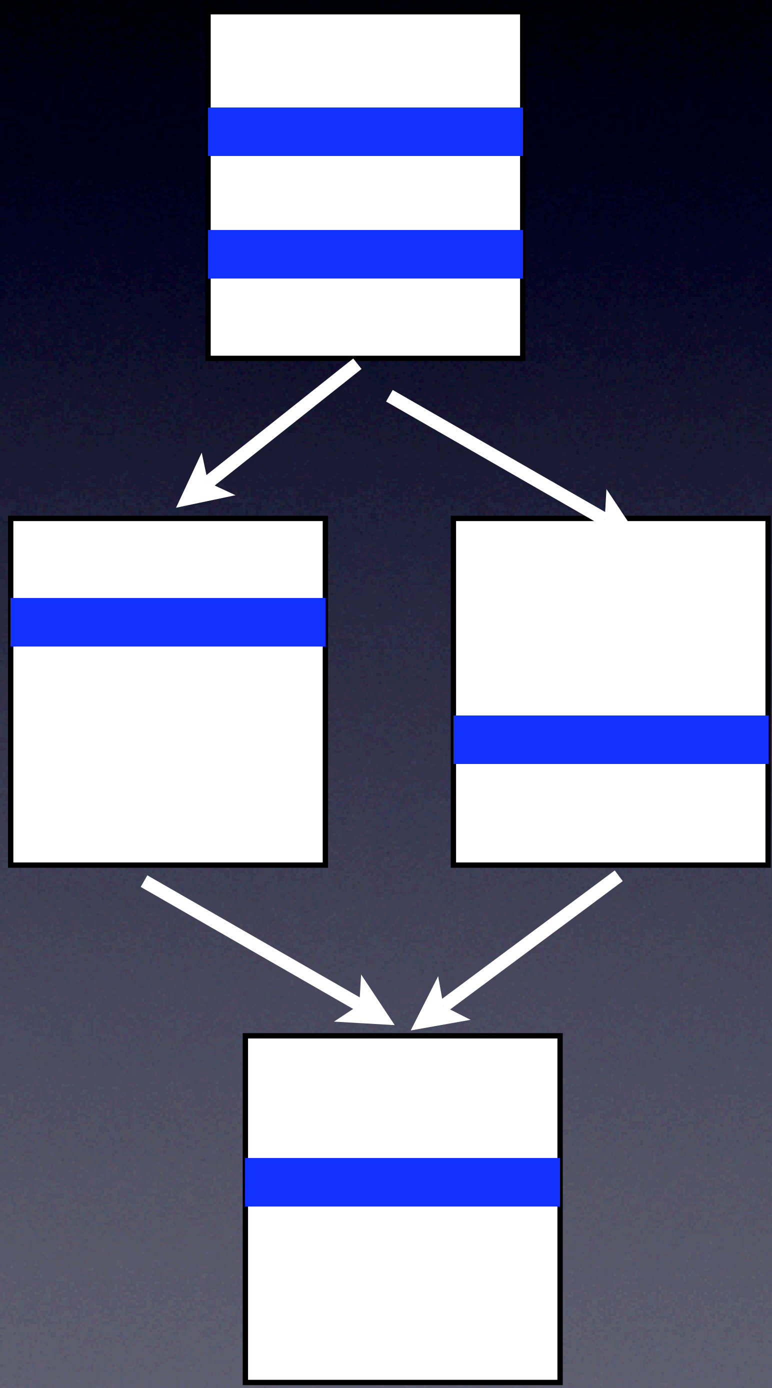


Simple Example

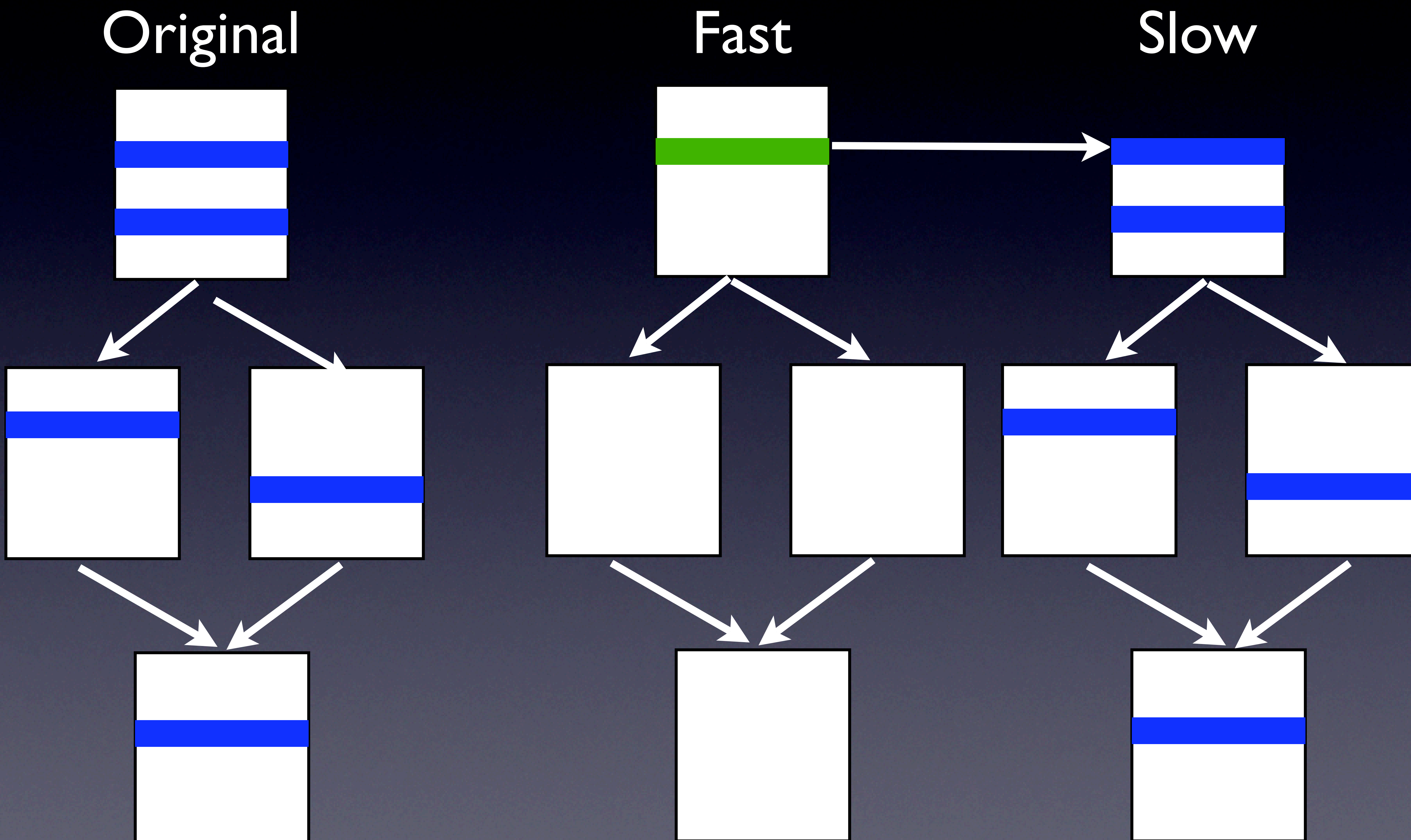


Simple Example

Original



Simple Example



How It Really Works

- We wish to provide best *throughput* while still being sound.
- Thus - we need to be able to allow code to switch between one version of the barrier to another when there is a phase change in the collector.
- This is the crucial difference from previous work on specialization.

GC points

- Typically, concurrent and incremental collectors require that each mutator acknowledges changes in phase at <u>GC points</u>.
- A GC point may be:
  - memory allocation
  - back branch (to ensure that GC points are reached in a timely fashion)
  - by proxy - any method call

How It Really Works

- Three versions of code:
  - Unspecialized - code where we don’t care about GC phase
  - Fast - code where we know that we don’t need barriers
  - Slow - code where we need barriers

- The approach:
  - The “Unspecialized” code is the original code; it will check phase, and switch to either Fast or Slow, at every barrier.
  - Fast and Slow switch to Unspecialized at GC points (e.g. method call).

A better example (Lazy Brooks)

```
int foo(object o) {
   int x = 2+2;
   o.f = x;
   o.g = null;
   o.bar();
   return o.f;
}
```

A better example (Lazy Brooks)

```
int foo(object o) {
    int x = 2+2;
    o.f = x;
    o.g = null;
    o.bar();
    return o.f;
}
```

Needs Barriers

Needs Barrier

A better example (Lazy Brooks)

```
int foo(object o) {
    int x = 2+2;
    o.f = x;          // Needs Barriers
    o.g = null;
    o.bar();          // GC point
    return o.f;       // Needs Barrier
}
```

Needs Barriers

GC point

Needs Barrier

Lazy Brooks: Without Specialization

```
int foo(object o) {
    int x = 2+2;
    o.forward.f = x;          // Needs Barriers
    o.forward.g = null;
    o.bar();                  // GC point
    return o.forward.f;       // Needs Barrier
}
```

Needs Barriers

GC point

Needs Barrier

What happens with path specialization?

```
int foo(object o) {
   int x = 2+2;
   o.f = x;
   o.g = null;
   o.bar();
   return o.f;
}
```

```
int foo(object o) {
  int x = 2+2;

  o.f = x;
  o.g = null;

  o.bar();

  return o.f;
}
```

Unspecialized

```
int foo(object o) {
  int x = 2+2;
    ↓
  o.f = x;
  o.g = null;
    ↓
  o.bar();
    ↓
  return o.f;
}
```

Fast

```
int foo(object o) {
  int x = 2+2;
    ↓
  o.f = x;
  o.g = null;
    ↓
  o.bar();
    ↓
  return o.f;
}
```

Slow

```
int foo(object o) {
  int x = 2+2;
    ↓
  o.forward.f = x;
  o.forward.g = null;
    ↓
  o.bar();
    ↓
  return o.forward.f;
}
```

Unspecialized	Fast	Slow
`int foo(object o) {` `int x = 2+2;`	`int foo(object o) {` `int x = 2+2;`	`int foo(object o) {` `int x = 2+2;`
`o.f = x;` `o.g = null;`	`o.f = x;` `o.g = null;`	`o.forward.f = x;` `o.forward.g = null;`
`o.bar();`	`o.bar();`	`o.bar();`
`return o.f;` `}`	`return o.f;` `}`	`return o.forward.f;` `}`

Unspecialized	Fast	Slow
int foo(object o) { int x = 2+2;	int foo(object o) {	int foo(object o) {
o.f = x; o.g = null;	o.f = x; o.g = null;	o.forward.f = x; o.forward.g = null;
o.bar();		
return o.f;	return o.f; }	return o.forward.f; }

Lazy Brooks: With Specialization

```
int foo(object o) {
    int x = 2+2;
    if need barrier o.forward.f = x;
                    o.forward.g = null;
    else o.f = x;
         o.g = null;
    o.bar();
    if need barrier return o.forward.f;
    else return o.f;
}
```

Lazy Brooks: With Specialization

Unspecialized

Unspecialized

```
int foo(object o) {
    int x = 2+2;
    if need barrier o.forward.f = x;
                    o.forward.g = null;
    else o.f = x;
         o.g = null;
    o.bar();
    if need barrier return o.forward.f;
    else return o.f;
}
```

Lazy Brooks: With Specialization

```
               int foo(object o) {
Unspecialized    int x = 2+2;
                 if need barrier o.forward.f = x;
                                 o.forward.g = null;
Fast             else o.f = x;
                      o.g = null;
Unspecialized    o.bar();
                 if need barrier return o.forward.f;
Fast             else return o.f;
               }
```

Lazy Brooks: With Specialization

	int foo(object o) {
Unspecialized	int x = 2+2;
Slow	if need barrier o.forward.f = x; o.forward.g = null;
Fast	else o.f = x; o.g = null;
Unspecialized	o.bar();
Slow	if need barrier return o.forward.f;
Fast	else return o.f;
	}

Summary

- Our algorithm aims to introduce the *smallest number of "needs barrier" phase checks along any path...*
- ... while ensuring that code is not duplicated unnecessarily (example: any path from a GC point to a check is not duplicated).
- *See the paper for the complete algorithm.*

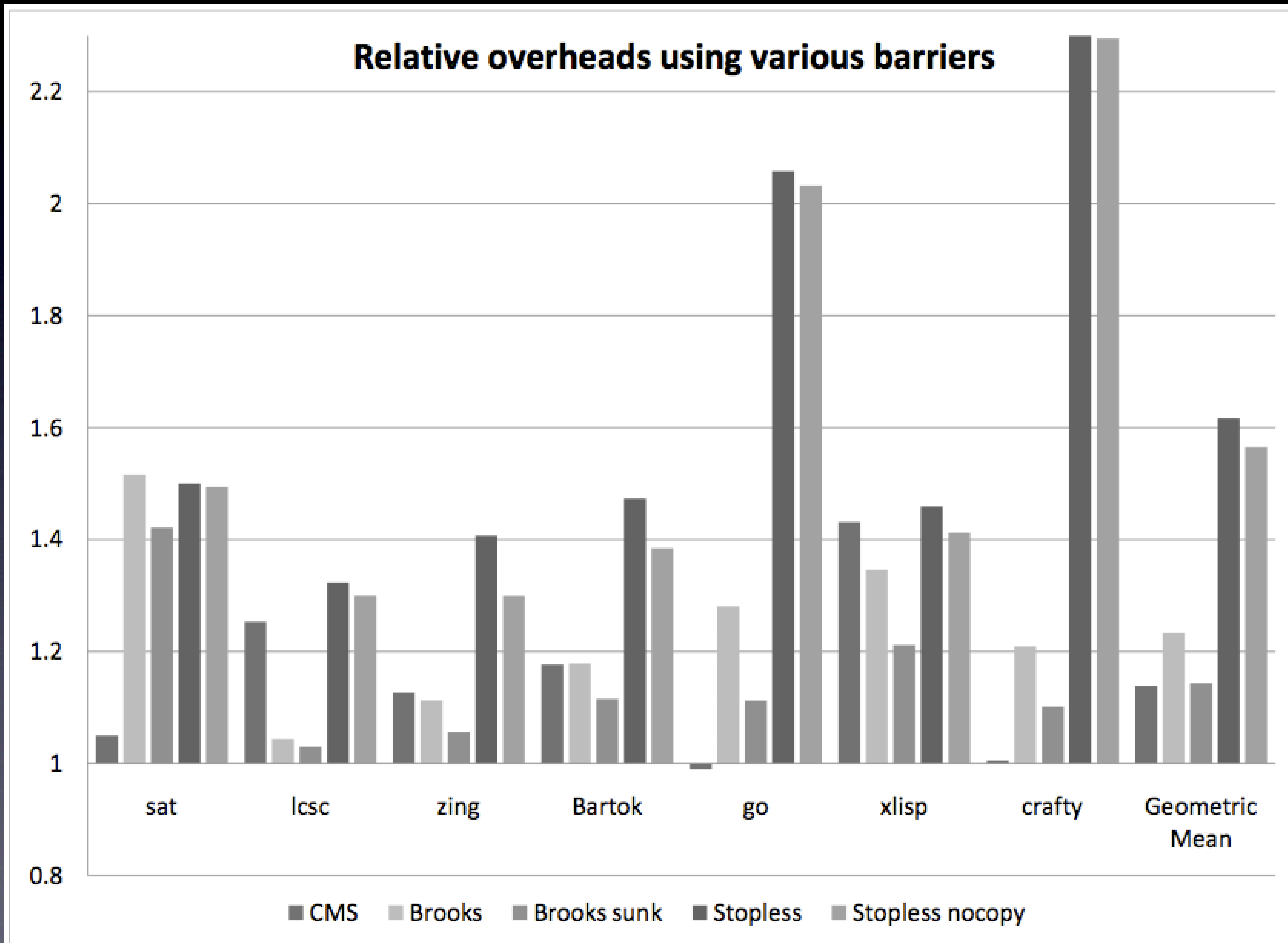
Implementation

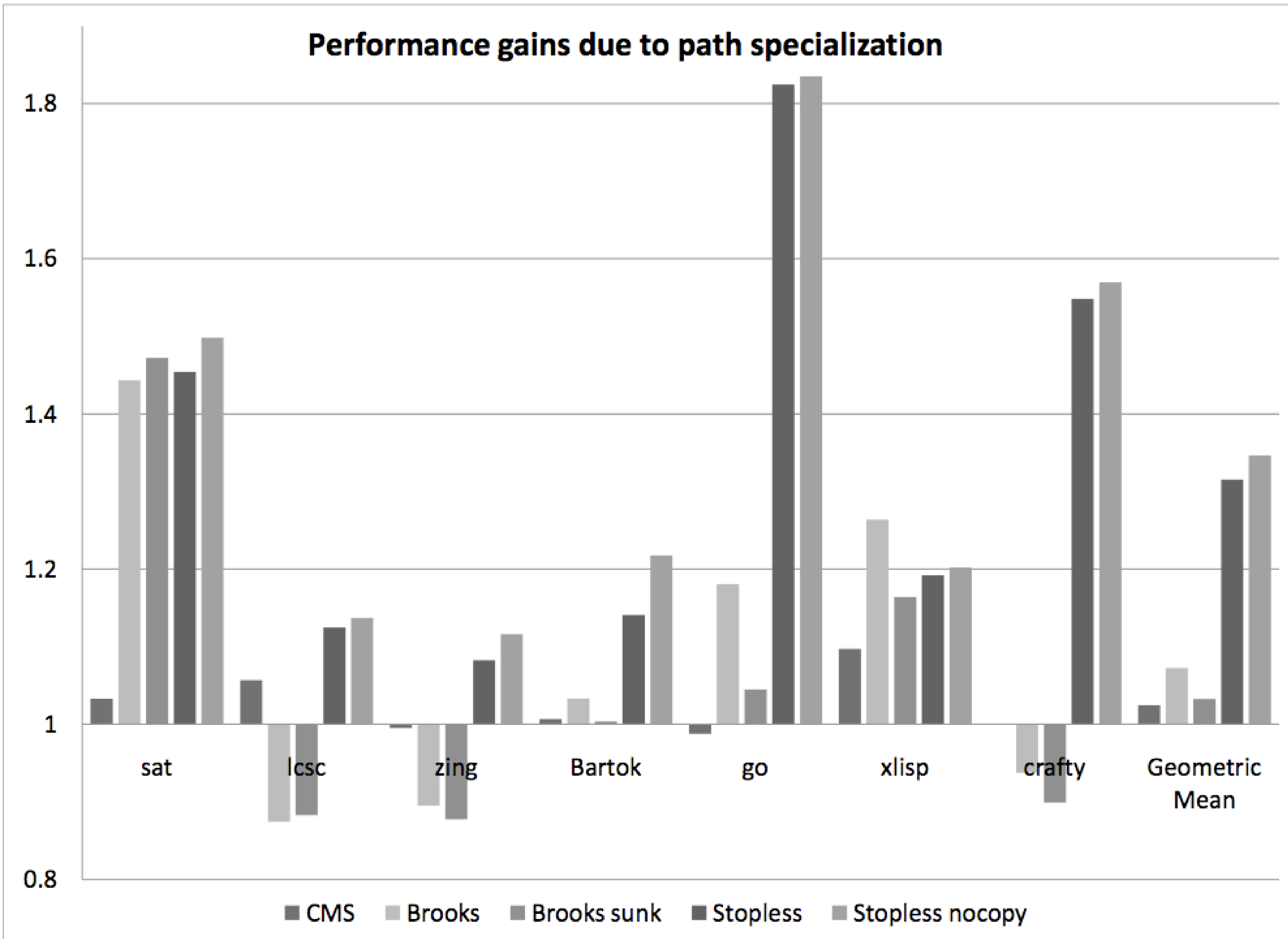
- We have implemented Path Specialization in the Microsoft Bartok Research Compiler.
- Path specialization exists as an optional pass that can be applied to any barrier that has a phase check.
- We have tested this with our Yuasa barrier, our lazy and eager Brooks barriers, and our Stopless barriers.

Results

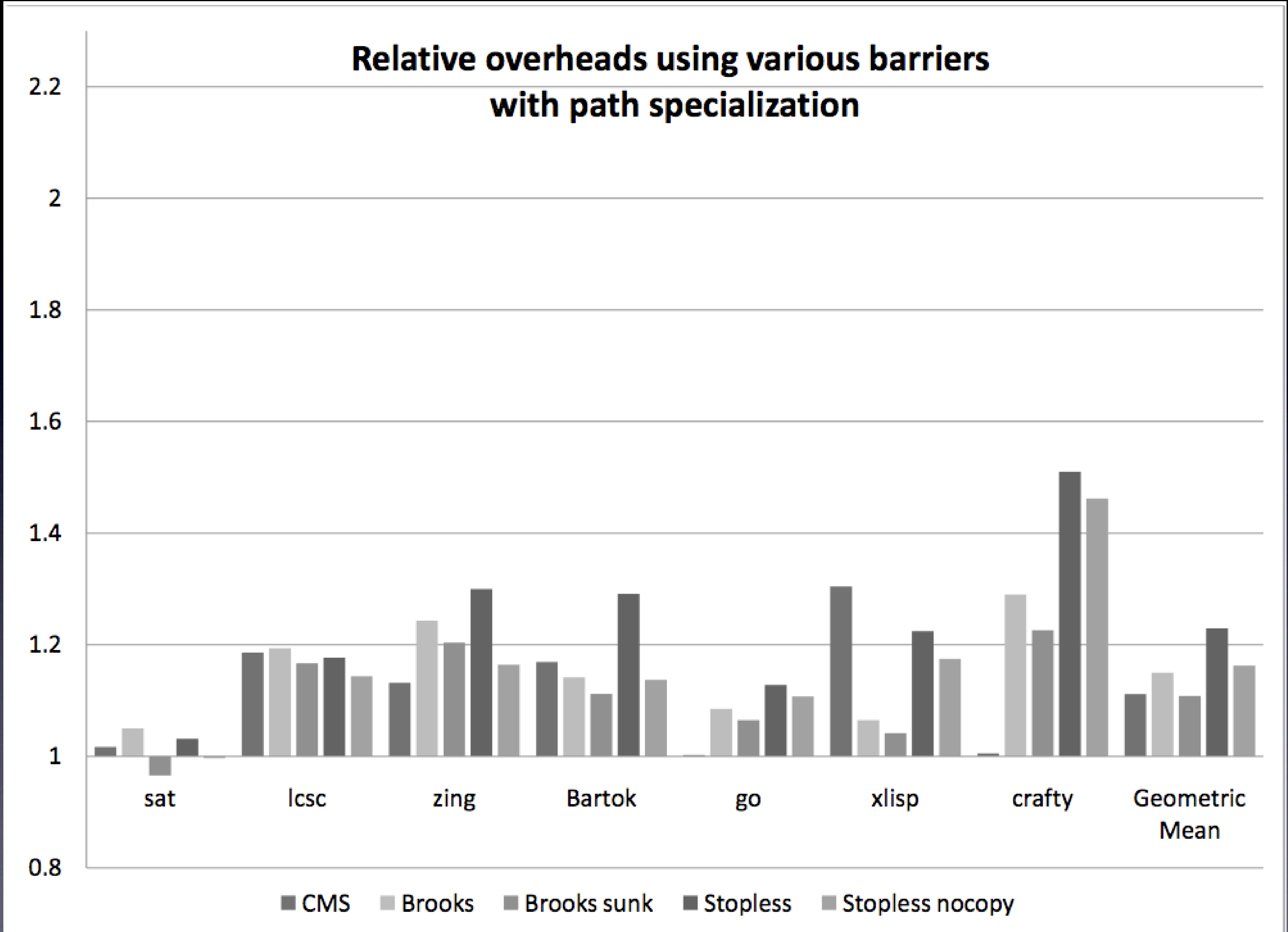
- We test four internal MSR benchmarks (large PL-type programs) and three smaller traditional benchmarks ported to .NET.
- Five barriers are used: CMS (Yuasa-type barrier), Brooks (lazy), Brooks (sunk eager), Stopless, and Stopless without any copying activity.

Without Specialization

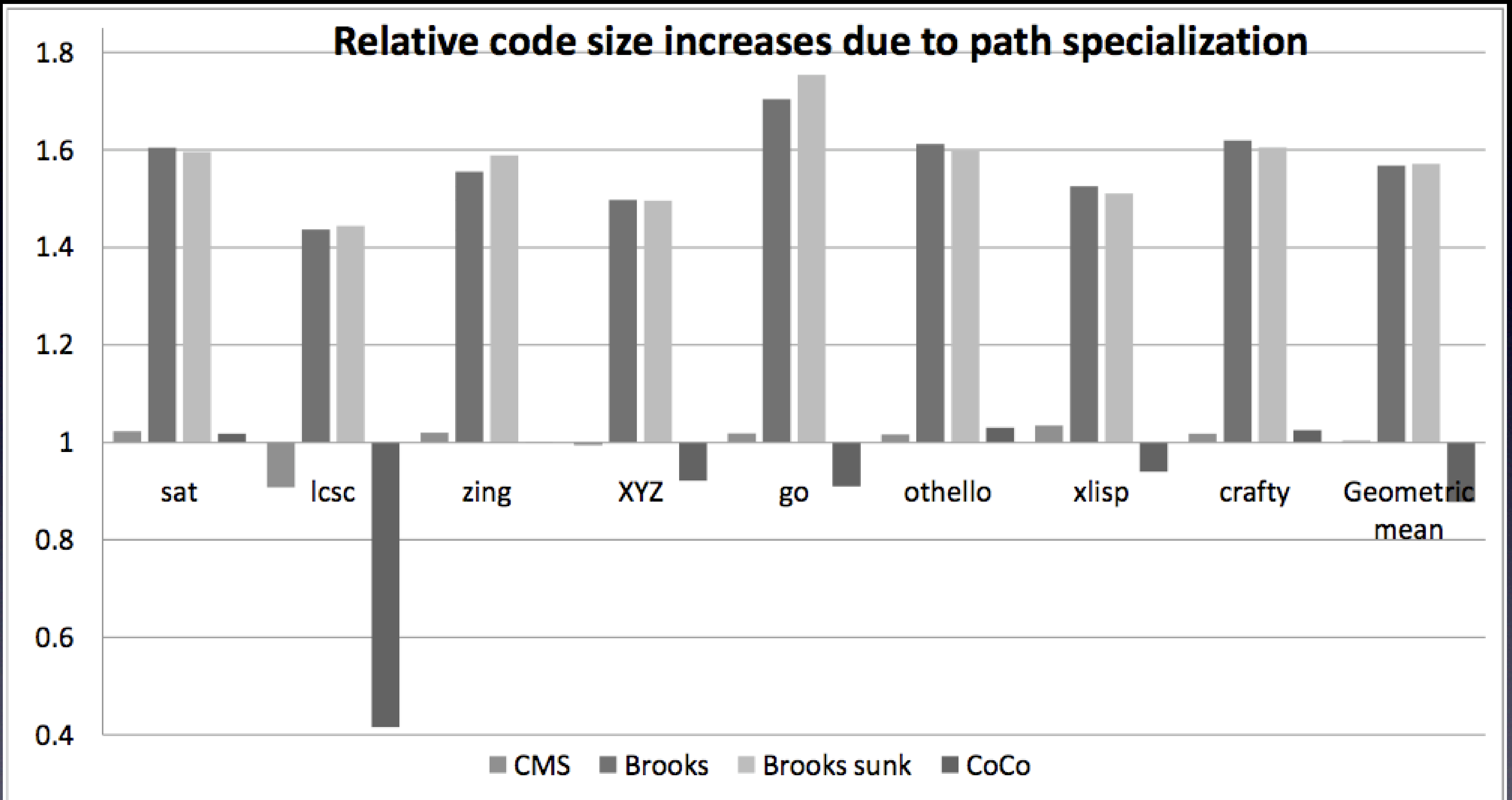




Performance gains due to path specialization
1.8
1.6
1.4
1.2
1
0.8
sat
lcsc
zing
Bartok
go
xlisp
crafty
Geometric Mean
CMS
Brooks
Brooks sunk
Stopless
Stopless nocopy



Relative overheads using various barriers
with path specialization
2.2
2
1.8
1.6
1.4
1.2
1
0.8
sat
lcsc
zing
Bartok
go
xlisp
crafty
Geometric Mean
CMS
Brooks
Brooks sunk
Stopless
Stopless nocopy


Relative code size increases due to path specialization
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
sat
lcsc
zing
XYZ
go
othello
xlisp
crafty
Geometric mean
CMS
Brooks
Brooks sunk
CoCo

Conclusion

- For heavy barriers (Stopless), path specialization reduces code size and improves performance.
- For barriers that are cheap but already have phase checks (like CMS), path specialization increases performance a bit without affecting code size.
- For Brooks barriers, performance improves but results in large code blow-up.
- <u>Performance improves for every barrier we tried.</u>

Questions/Comments