Concurrency Analysis for Parallel Programs with Textually Aligned Barriers

Amir Kamil and Katherine Yelick
Titanium Group
http://titanium.cs.berkeley.edu

U.C. Berkeley
October 20, 2005

Motivation/Goals

- **Many program analyses and optimizations benefit from knowing which expressions can run concurrently**
- **Develop basic concurrency analysis for Titanium programs**
- **Refine analysis to ignore *infeasible* paths**
- **Evaluate analysis using two applications**
  - Race detection
  - Memory model enforcement

Barrier Alignment

- **Many parallel languages make no attempt to ensure that barriers line up**
  - Example code that is legal but will deadlock:

```
if (Ti.thisProc() % 2 == 0)
  Ti.barrier(); // even ID threads
else
  ; // odd ID threads
```

Structural Correctness

- **Aiken and Gay introduced *structural correctness (POPL'98)***
  - Ensures that every thread executes the same number of barriers
  - Example of structurally correct code:

```
if (Ti.thisProc() % 2 == 0)
  Ti.barrier(); // even ID threads
else
  Ti.barrier(); // odd ID threads
```

Textual Barrier Alignment

- **Titanium has *textual barriers*: all threads must execute the same *textual* sequence of barriers**
  - Stronger guarantee than structural correctness – this example is illegal:

```
if (Ti.thisProc() % 2 == 0)
   Ti.barrier(); // even ID threads
else
   Ti.barrier(); // odd ID threads
```

- ***Single-valued* expressions used to enforce textual barriers**

Single-Valued Expressions

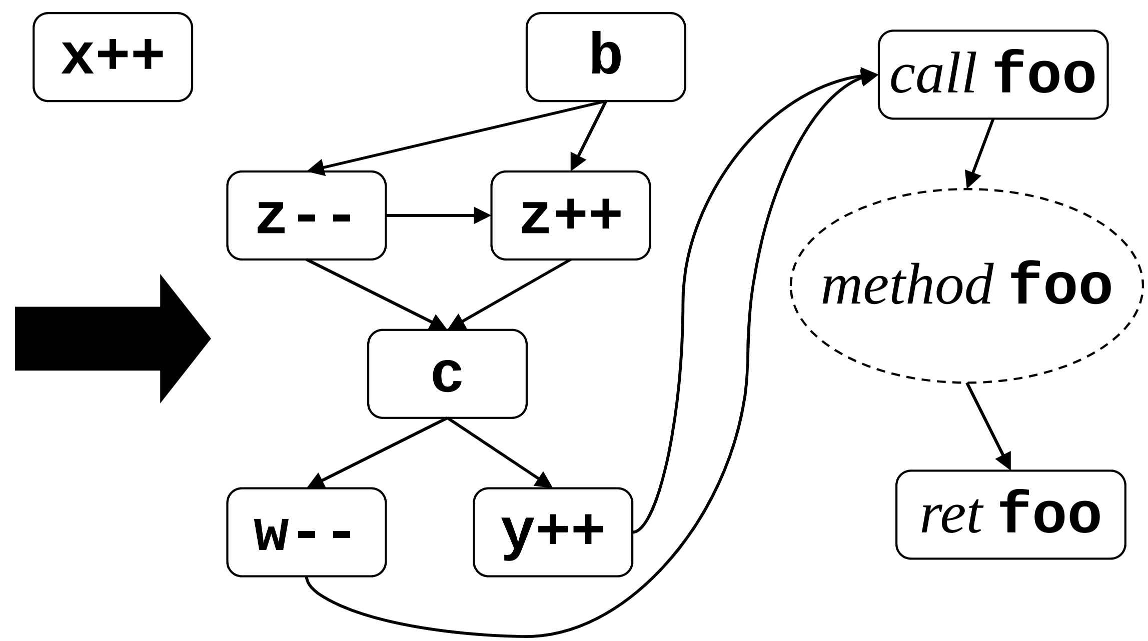
- **A *single-valued* expression has the same value on all threads when evaluated**
  - Example: **`Ti.numProcs() > 1`**
- **All threads guaranteed to take the same branch of a conditional guarded by a single-valued expression**
  - Only single-valued conditionals may have barriers
  - Example of legal barrier use:

```
if (Ti.numProcs() > 1)
   Ti.barrier(); // multiple threads
else
   ; // only one thread
```

Concurrency Graph

- **Represents concurrency as a graph**
  - Nodes are program expressions
  - If a path exists between two nodes, they can run concurrently
- **Generated from control flow graph**

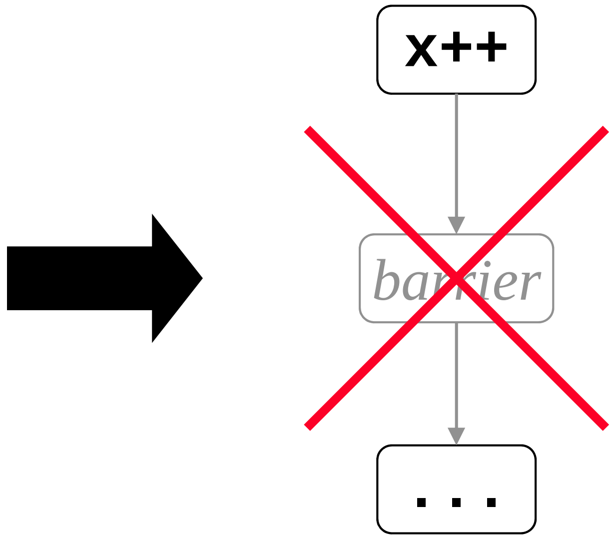
```
x++;
Ti.barrier();
if (b) z--;
else z++;
if (c [single]) w--;
else y++;
foo();
```



Barriers

- Barriers prevent code before and after from running concurrently
- Nodes for barrier expressions removed from concurrency graph

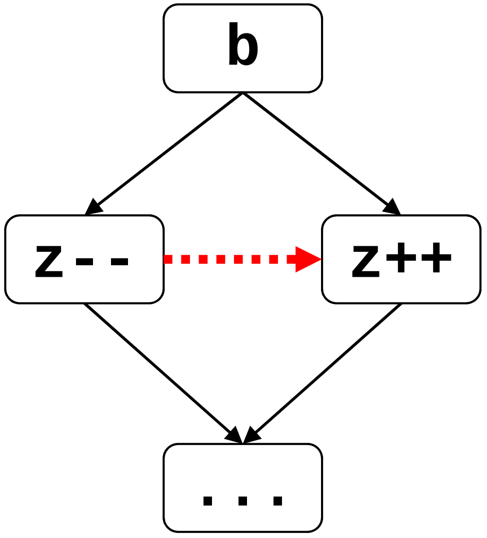
```
x++;
Ti.barrier();
...
```



Non-Single Conditionals

- **Branches of a non-single conditional can run concurrently**
  - Different threads can take different branches
- **Edge added in the concurrency graph between branches**

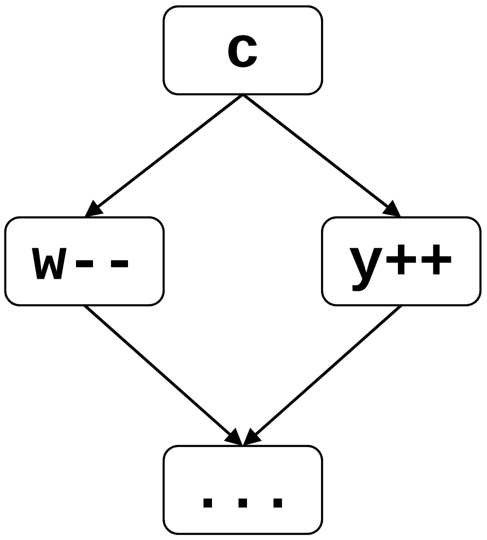
```
if (b)
  z--;
else
  z++;
...
```



Single Conditionals

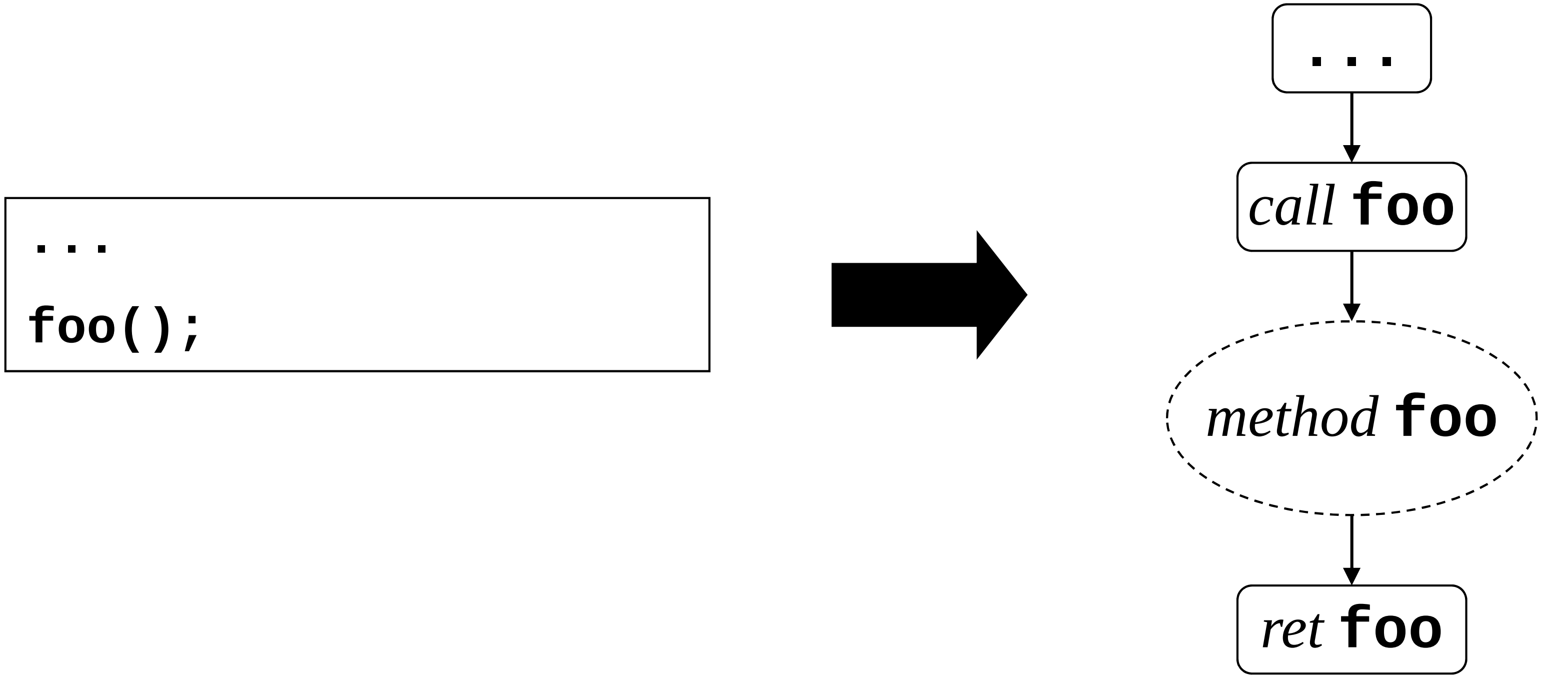
- **Branches of a single conditional cannot run concurrently**
  - All threads take the same branch
- **No edge added in the concurrency graph between branches**

```
if (c [single])
  w--;
else
  y++;
...
```



Method Calls

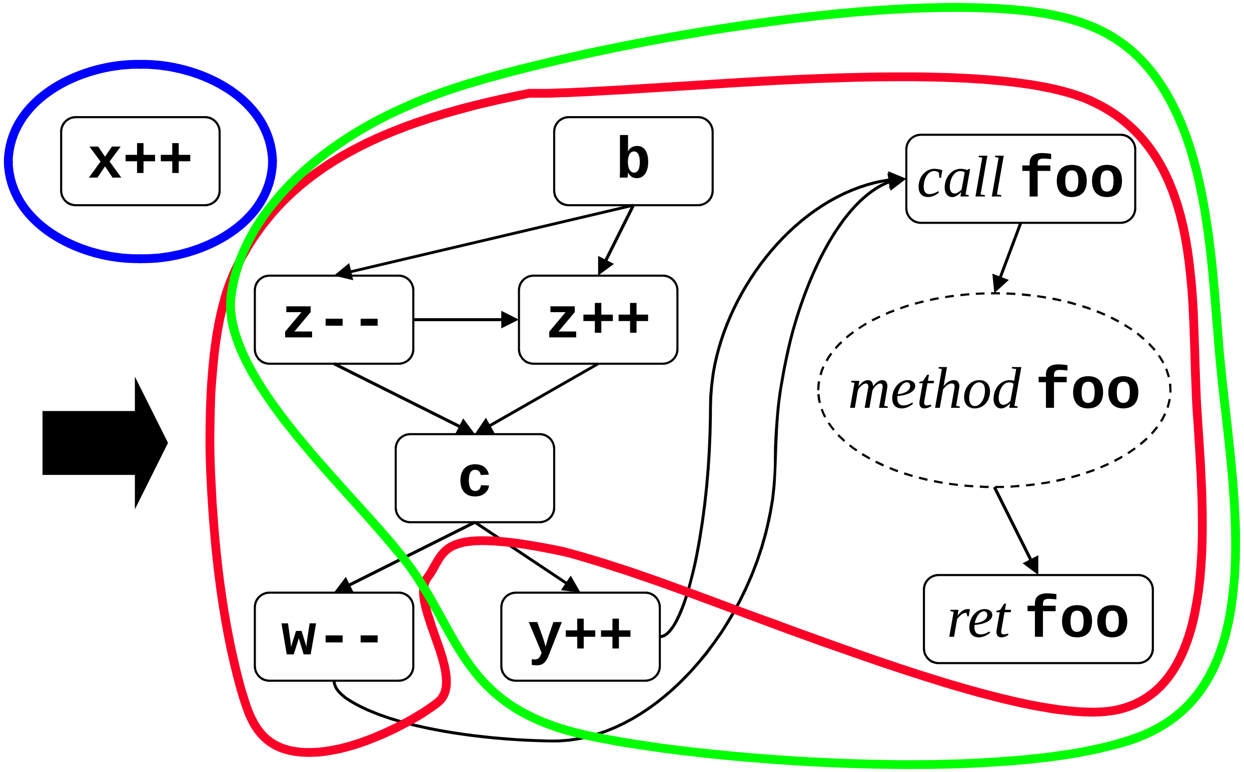
- **Method call nodes split into a call and return node**
  - Edges added from call node to target method's subgraph, and from target method to return node



Concurrency Algorithm

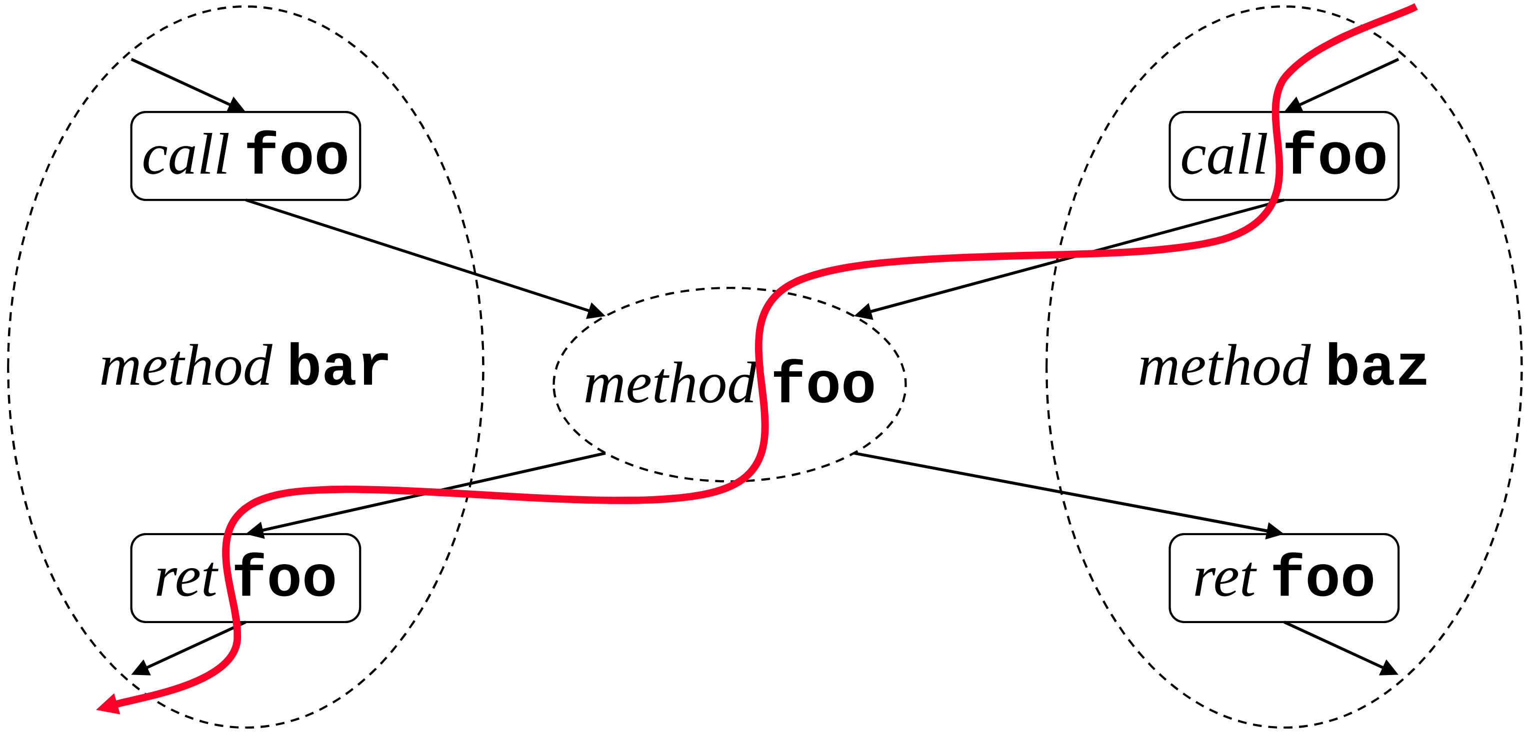
- **Two accesses can run concurrently if at least one is reachable from the other**
- **Concurrent accesses computed by doing *N* depth first searches**

```
x++;
Ti.barrier();
if (b) z--;
else z++;
if (c [single]) w--;
else y++;
foo();
```



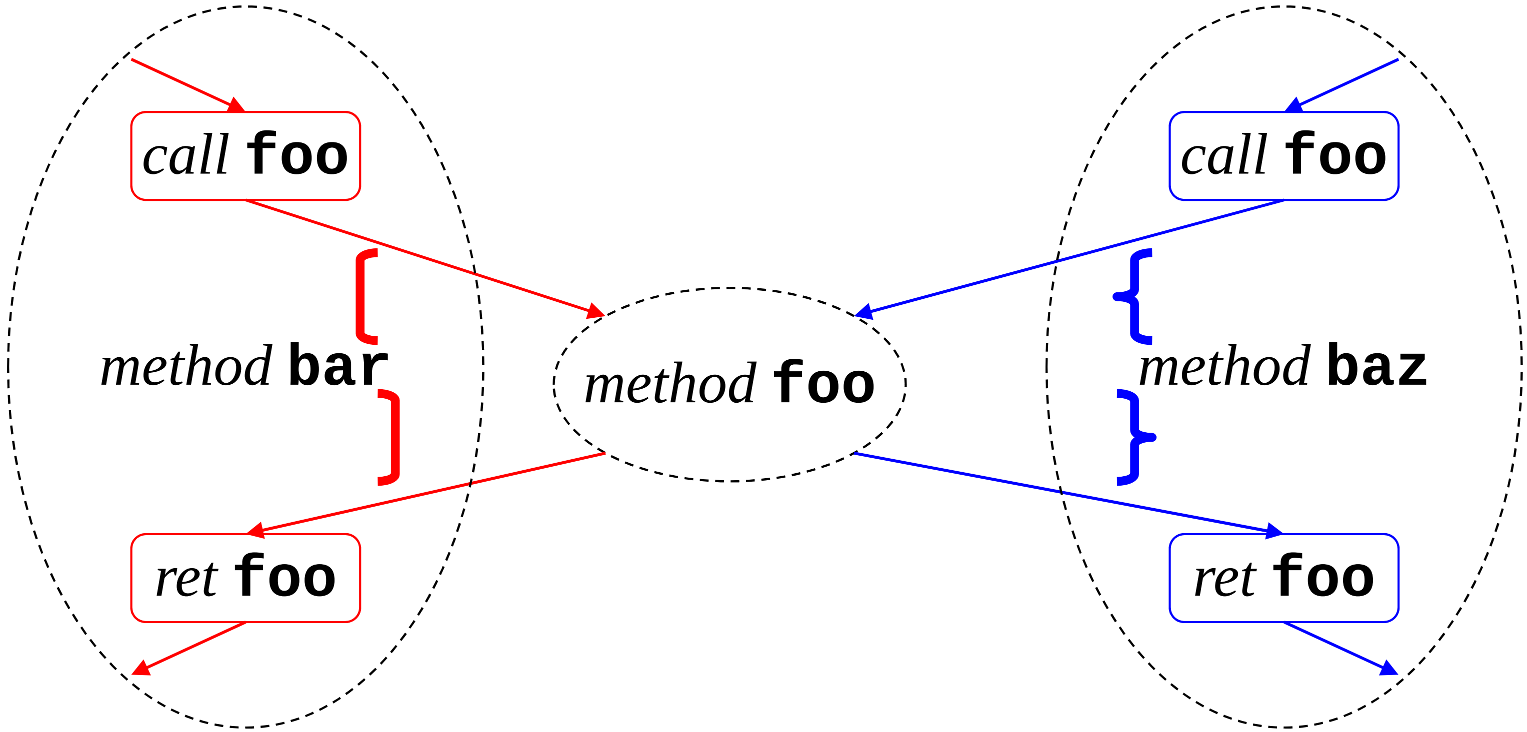
Infeasible Paths (I)

- **Handling of method calls allows *infeasible* control flow paths**
  - Path exists into one call site and out of another



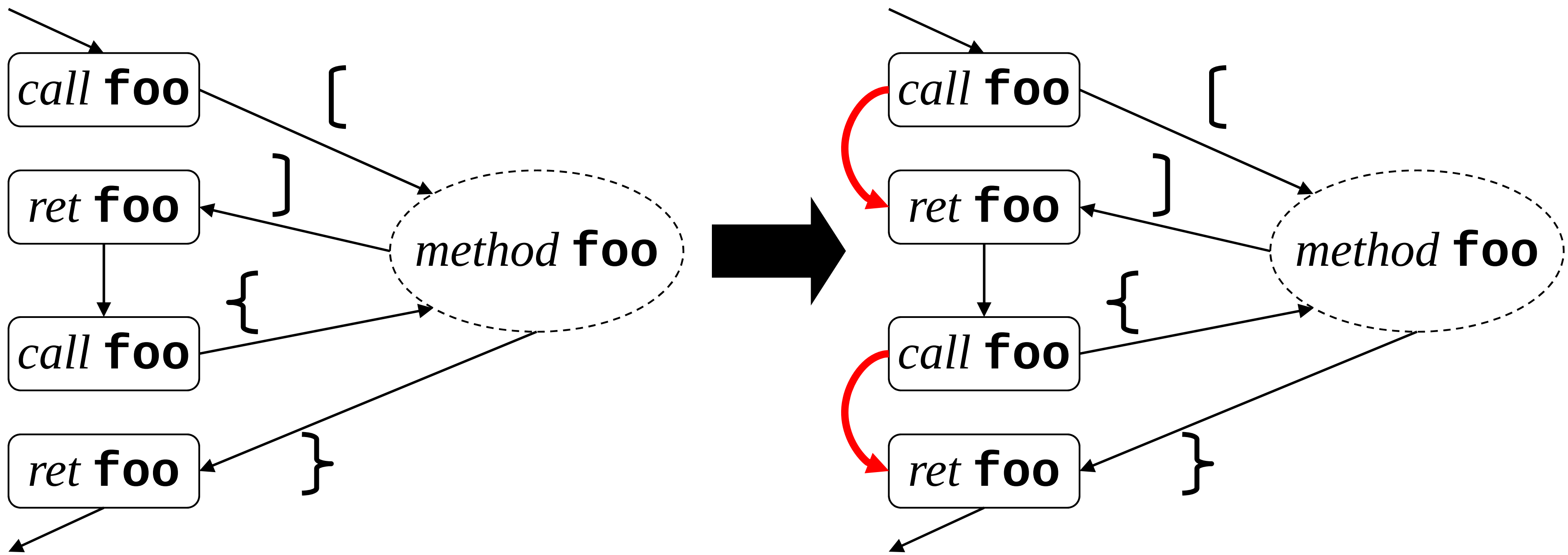
Infeasible Paths (II)

- **Solution – label call and return edges with matching parentheses**
  - Only follow paths that correspond to balanced parentheses



Bypass Edges (I)

- Reachability now depends on context
- Inefficient to revisit method in every context
- Solution – add edges to bypass method calls



Bypass Edges (II)

- **Can only bypass method calls that can actually complete (without executing a barrier)**
- **Iteratively compute set of methods that can complete**

> *CanComplete* ← φ
> Do (until a fixed point is reached):
> *CanComplete* ← *CanComplete* ∪ all methods that can complete by only calling methods in *CanComplete*

Static Race Detection

- **Two heap accesses compose a *data race* if they can concurrently access the same location, and at least one is a write**

Initially, `x = 0`

T1	T2
set `x = x + 1`	set `x = x - 1`

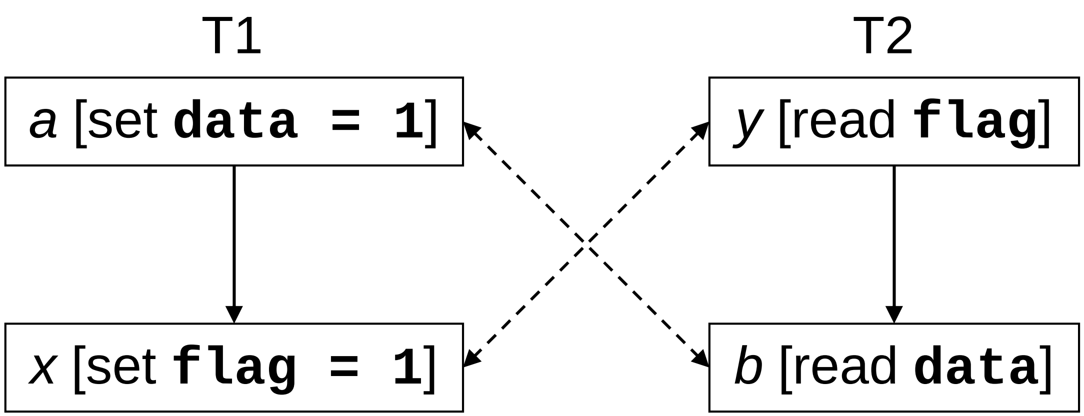
Possible final values of `x`:
-1, 0, 1

- **Alias and concurrency analysis used to statically compute set of possible data races**
  - Analyses are sound, so all real races are detected
- **Goal is to minimize number of false races detected**

Sequential Consistency

Definition: A parallel execution must behave as if it were an interleaving of the serial executions by individual threads, with each individual execution sequence preserving the program order [Lamport79].

Initially, **`flag = data = 0`**



Legal execution: a x y b

Illegal execution: x y b a

Critical cycle

Titanium and most other languages ***do not*** provide sequential consistency due to the (perceived) cost of enforcing it.

Enforcing Sequential Consistency

- **Compiler and architecture must not reorder memory accesses that are part of a critical cycle**
- **Fences inserted into program to enforce order**
  - Potentially costly – can prevent optimizations such as code motion and communication aggregation
  - At runtime, can cost an RTT on a distributed machine
- **Goal is to minimize number of inserted fences**

Benchmarks

Benchmark	Lines[1]	Description
`lu-fact`	420	Dense linear algebra
`gsrb`	1090	Computational fluid dynamics kernel
`spmv`	1493	Sparse matrix-vector multiply
`pps`	3673	Parallel Poisson equation solver
`gas`	8841	Hyperbolic solver for gas dynamics

[1] Line counts do not include the reachable portion of the 37,000 line Titanium/Java 1.0 libraries

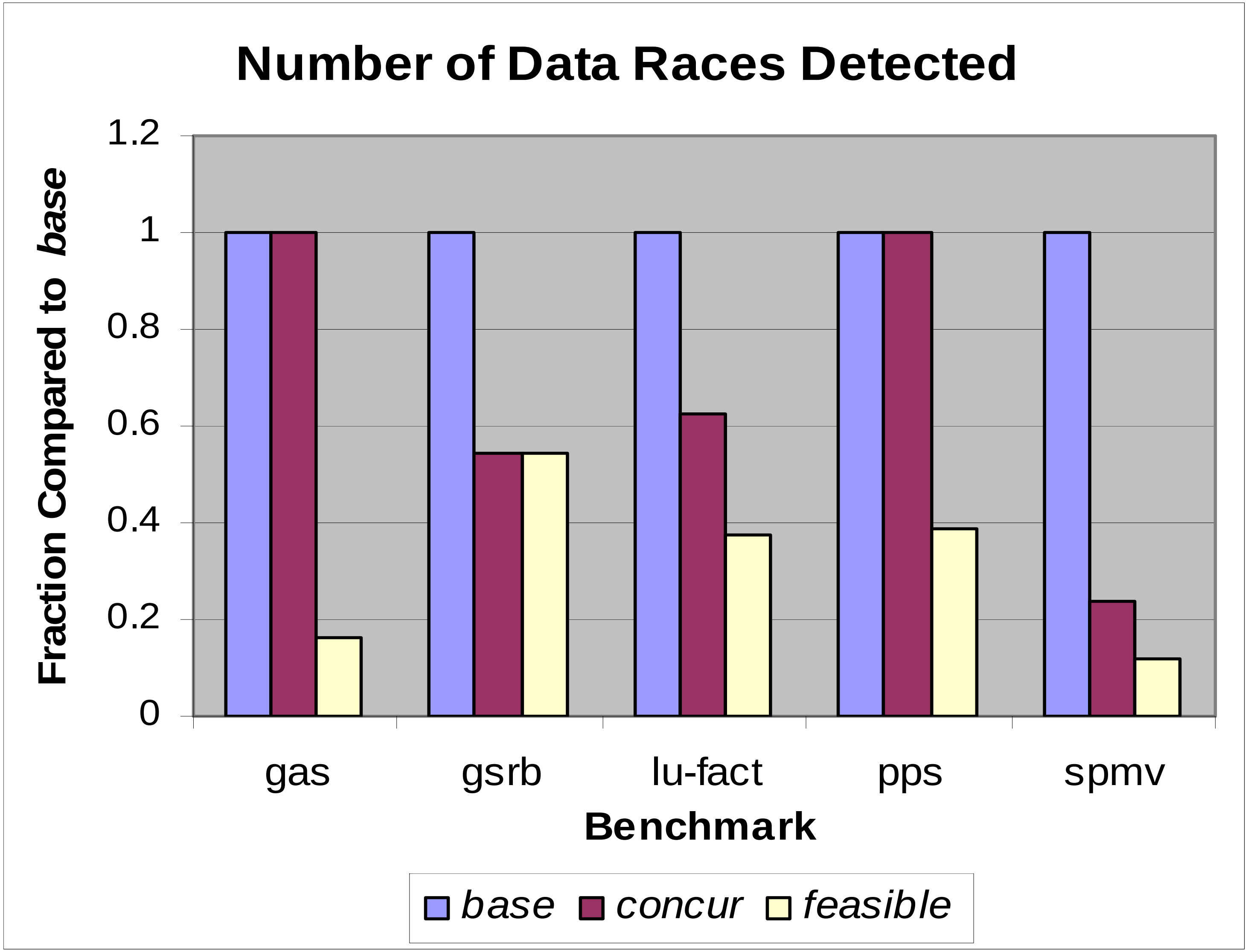
Analysis Levels

- **We tested analyses of varying levels of precision**

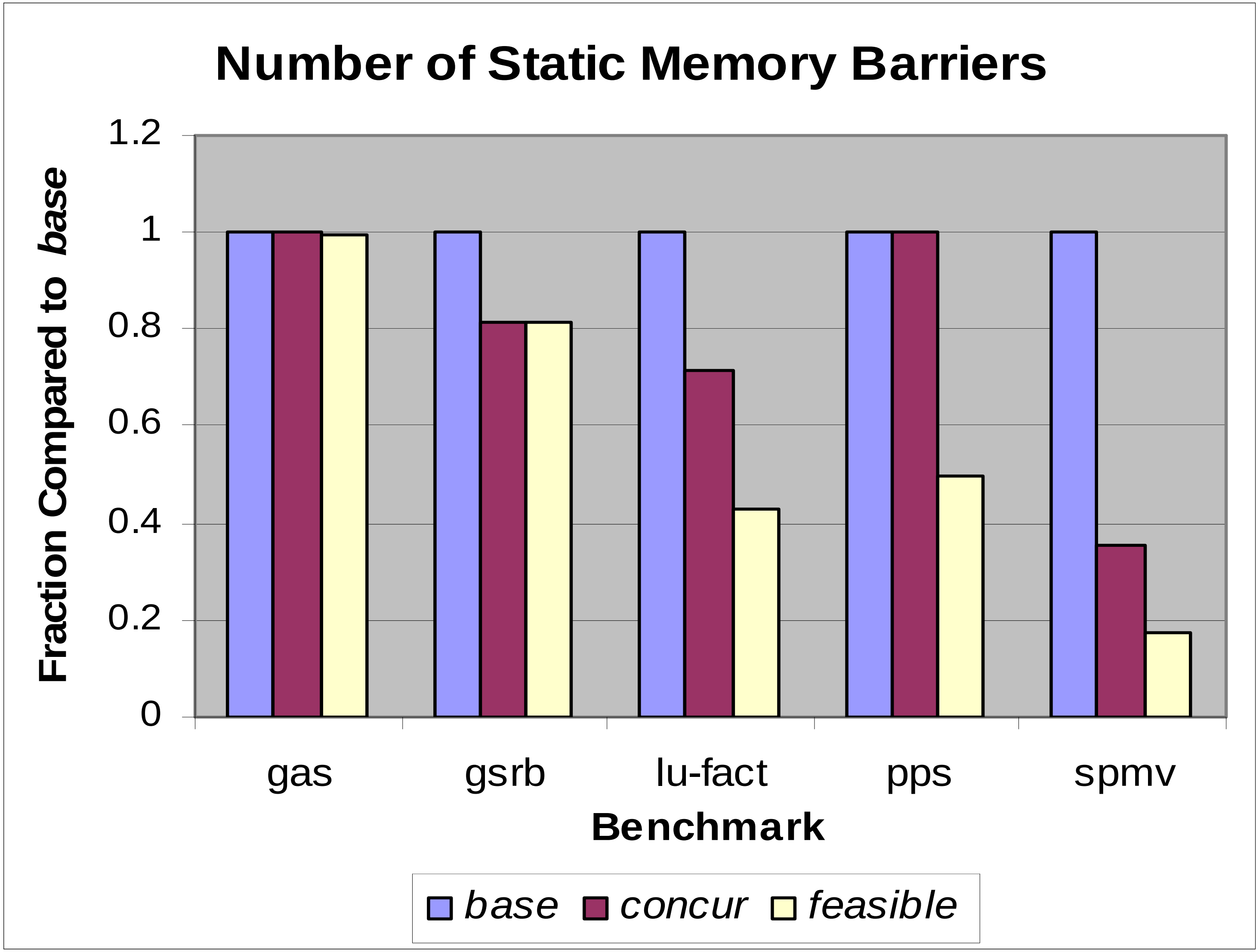
Analysis	**Description**
`base`	**All expressions assumed concurrent**
`concur`	**Basic concurrency analysis**
`feasible`	**Feasible paths concurrency analysis**

- **All levels use alias analysis to distinguish memory locations**

Race Detection Results



Sequential Consistency Results



Conclusion

- **Textual barriers and single-valued expressions allow for simple but precise concurrency analysis**
- **Concurrency analysis is useful both for detecting races and for enforcing sequential consistency**
  - Not sufficient for race detection – too many false positives
  - Good enough for sequential consistency to be provided at low cost *(SC|05)*
- **Ignoring infeasible paths significantly improves analysis results**