EECS 583 – Class 18
Research Topic 1
Breaking Dependences, Dynamic Parallelization

University of Michigan

November 21, 2011

Announcements & Reading Material

- No class on Wednes (no paper summary either!)
- We are grading the exams
- Today's class reading
  - "Spice: Speculative Parallel Iteration Chunk Execution," E. Raman, N. Vachharajani, R. Rangan, and D. I. August, *Proc 2008 Intl. Symposium on Code Generation and Optimization*, April 2008.
- Next class reading (Monday, Nov 28)
  - "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," M. Gordon, W. Thies, and S. Amarasinghe, *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

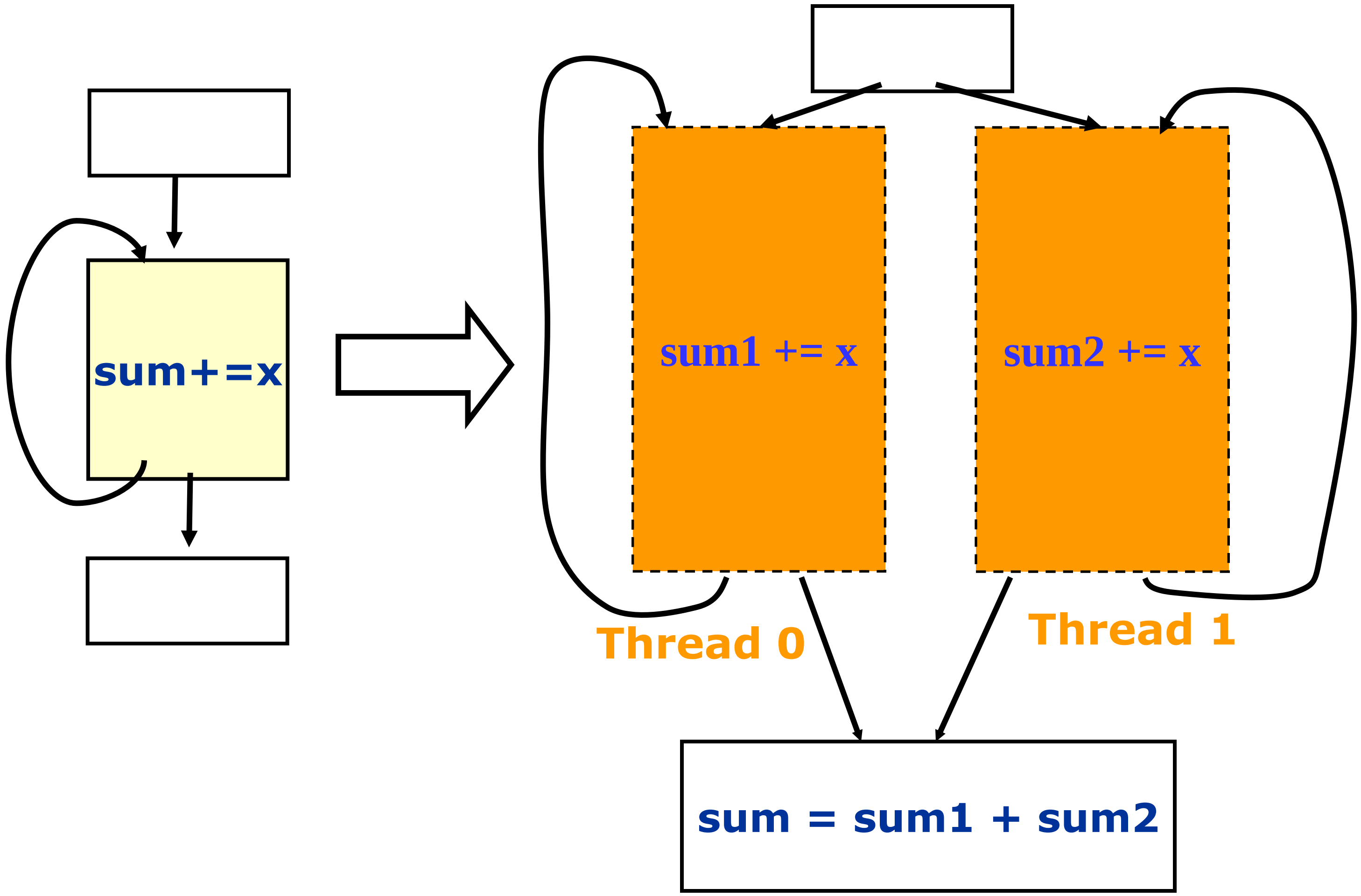
The Data Dependence Problem

```
while (ptr != NULL) {
    . . .
    ptr = ptr->next;
    sum = sum + foo;
}
```

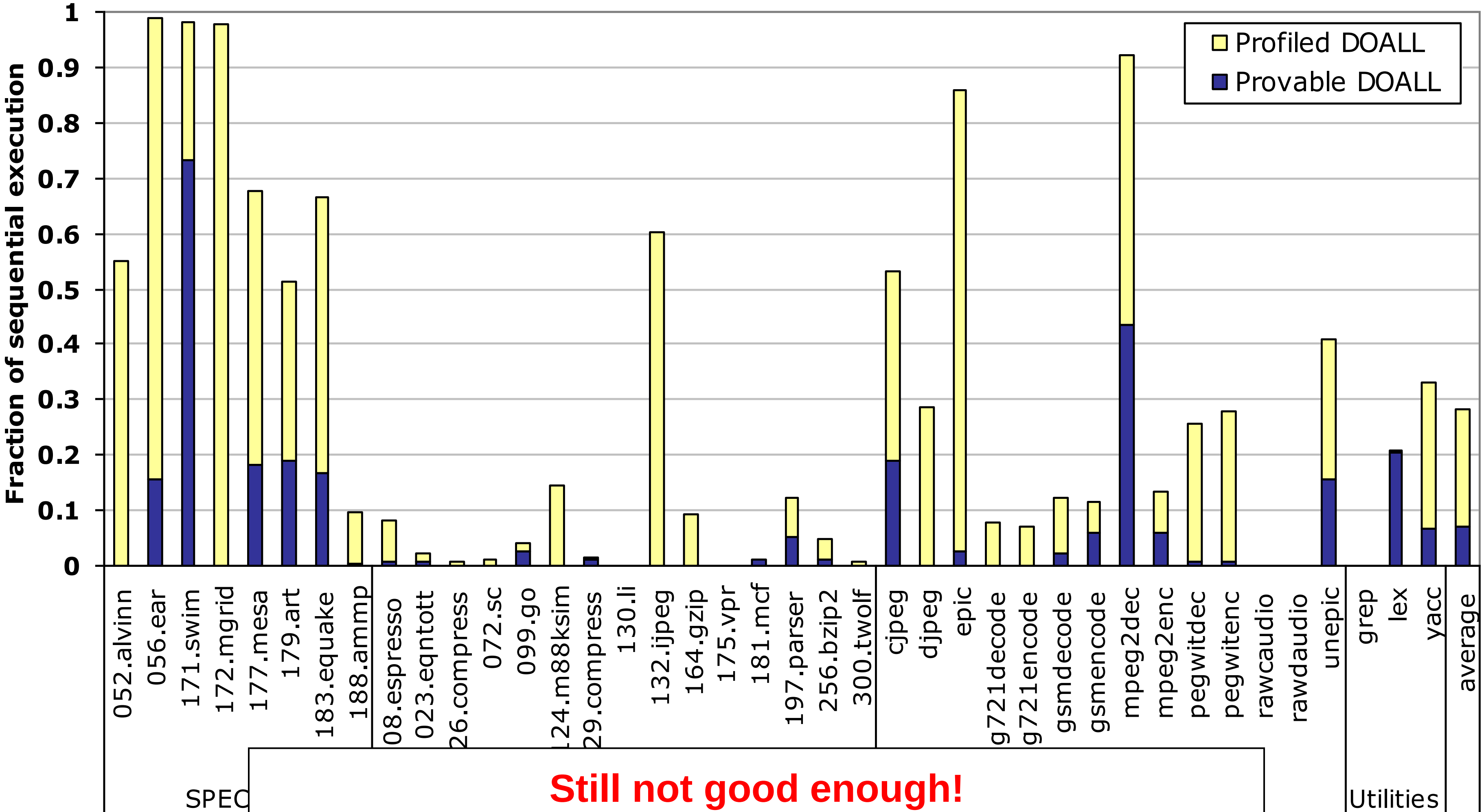
- How to deal with control dependences?
- How to deal with linked data structures?
- How to remove programmatic dependences?

We Know How to Break Some of These Dependences – Recall ILP Optimizations

Apply accumulator variable expansion!



DOALL Coverage – Provable and Profiled



Still not good enough!

Few dependences hinder parallelization in many loops

Speculative Loop Fission

```
1: while (node) {
2:   work(node);
3:   node = node->next;
   }
```

Sequential part

```
1: while (node) {
4:   node_array[count++] = node;
3:   node = node->next;
   }
```

Parallel part

```
   XBEGIN
5: node = node_array[IS];
   i = 0;
1':while (node && i++ < CS) {
2:    work(node);
3':   node = node->next;
   }
   RECV(THREAD_{j-1})
   XCOMMIT
   SEND(THREAD_{j+1})
   }
```

If this were traversing an array, it would be a DOALL loop

Separate out data structure access and work

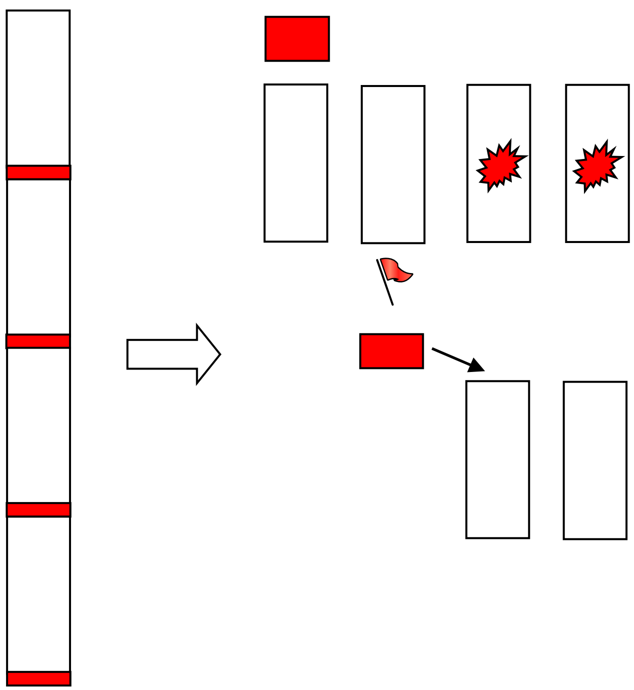
Parallelize work

Execution of Fissed Loop

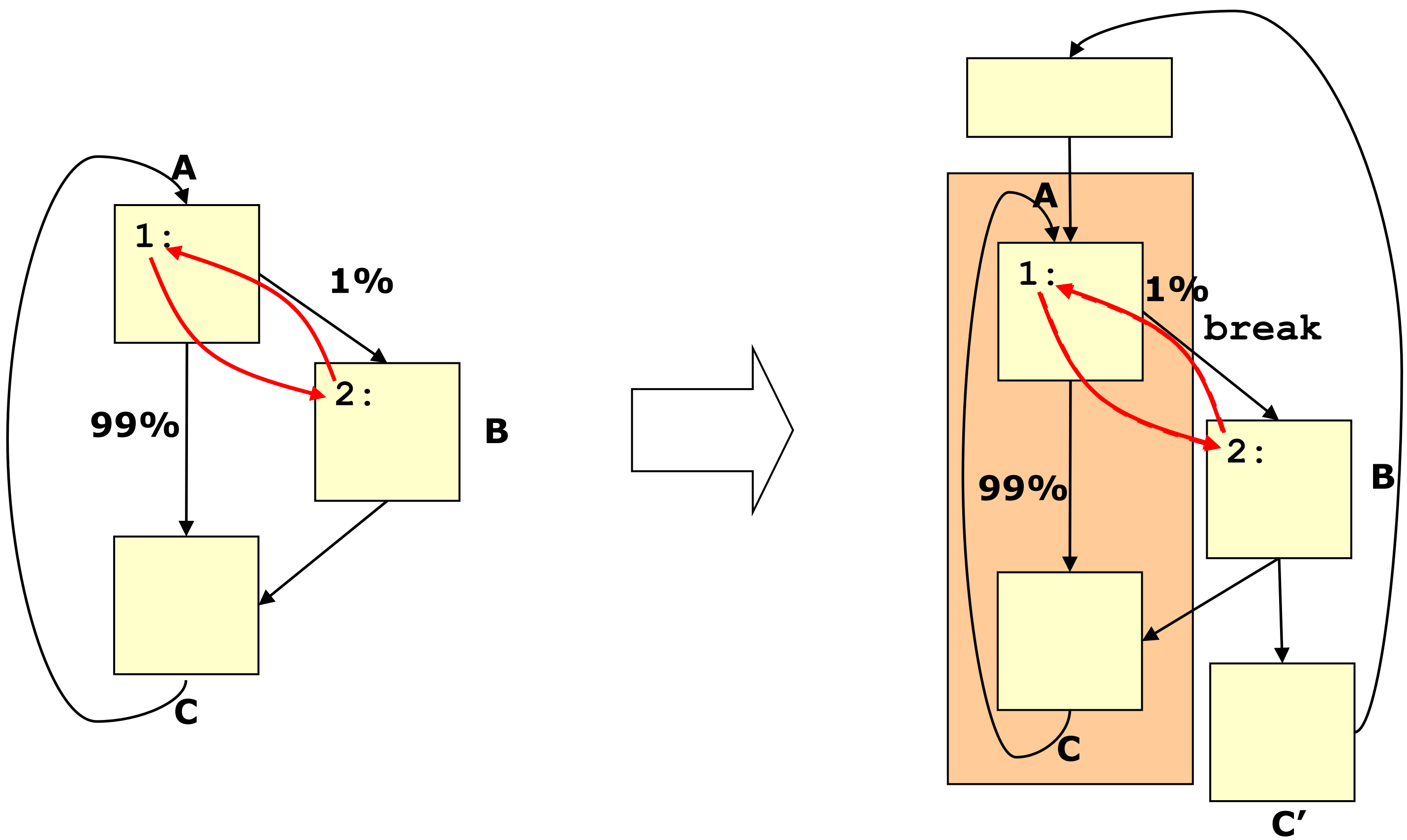
```
1: while (node) {
2:   work(node);
3:   node = node->next;
   }
```

```
1: while (node) {
4:   node_array[count++] = node;
3:   node = node->next;
   }
```

```
   XBEGIN
5: node = node_array[IS];
   i = 0;
1':while (node && i++ < CS) {
2:     work(node);
3':    node = node->next;
     }
   RECV(THREAD_{j-1})
   XCOMMIT
   SEND(THREAD_{j+1})
   }
```



Infrequent Dependence Isolation



Infrequent Dependence Isolation - cont

```
for( j=0; j<=nstate; ++j ){
 if( tystate[j] == 0 ) continue;
 if( tystate[j] == best ) continue;
 count = 0;
 cbest = tystate[j];
 for (k=j; k<=nstate; ++k)
  if (tystate[k]==cbest) ++count;
 if ( count > times) {
  best = cbest;
  times = count;
 }
}
```

1 %

Sample loop from yacc benchmark

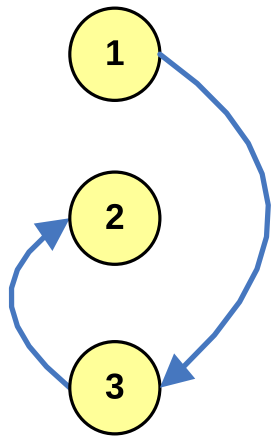
```
 j=0;
while (j<=nstate){
 for( ; j<=nstate; ++j ){
  if( tystate[j] == 0 ) continue;
  if( tystate[j] == best ) continue;
  count = 0;
  cbest = tystate[j];
  for (k=j; k<=nstate; ++k)
   if (tystate[k]==cbest) ++count;
  if ( count > times)
   break;
 }
 if (count > times) {
  best = cbest;
  times = count; j++;
 }
}
```

1 %

1 %

Speculative Prematerialization

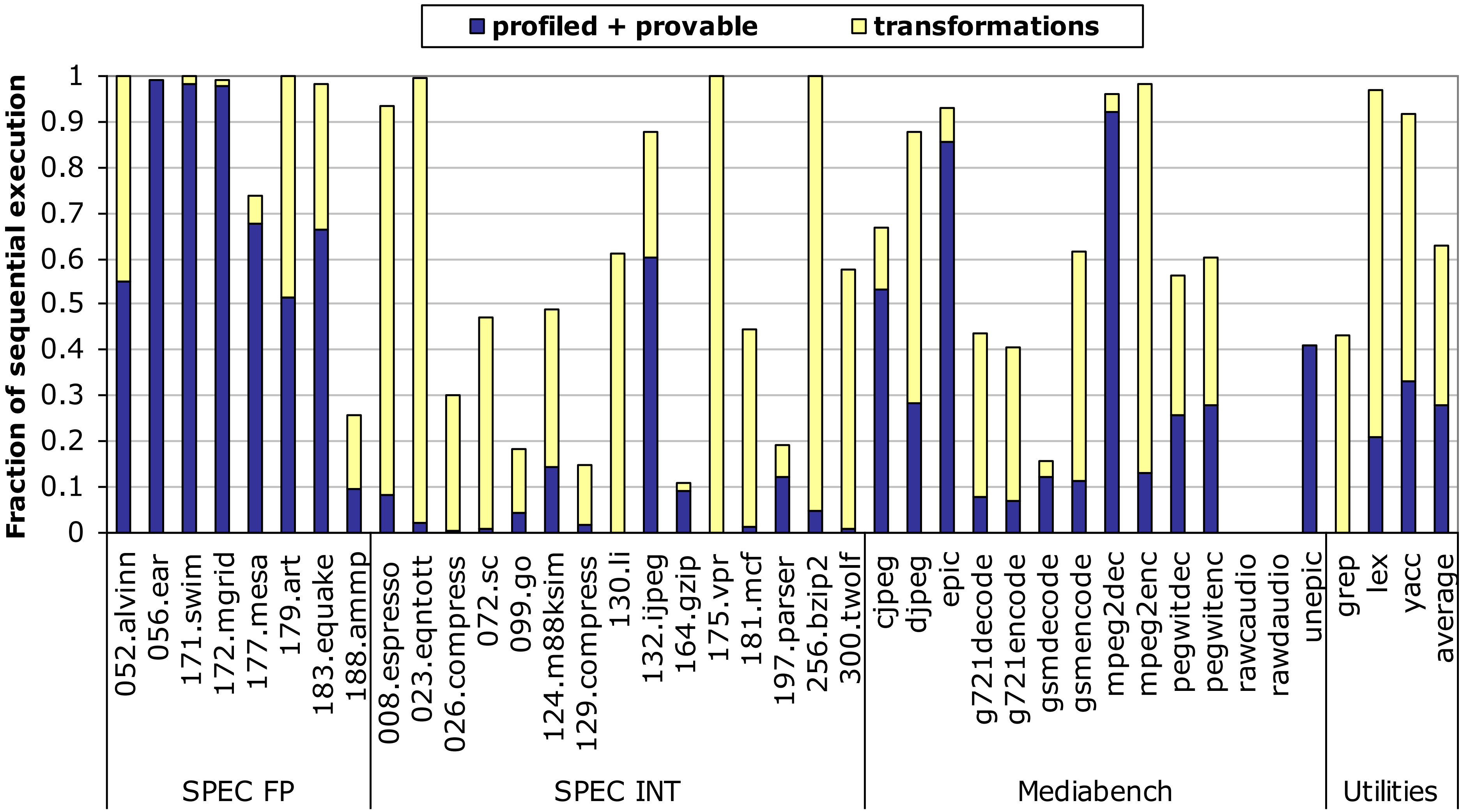
```
   for (...) {
1:   current = … ;
2:   work(last);
3:   last = current;
   }
```



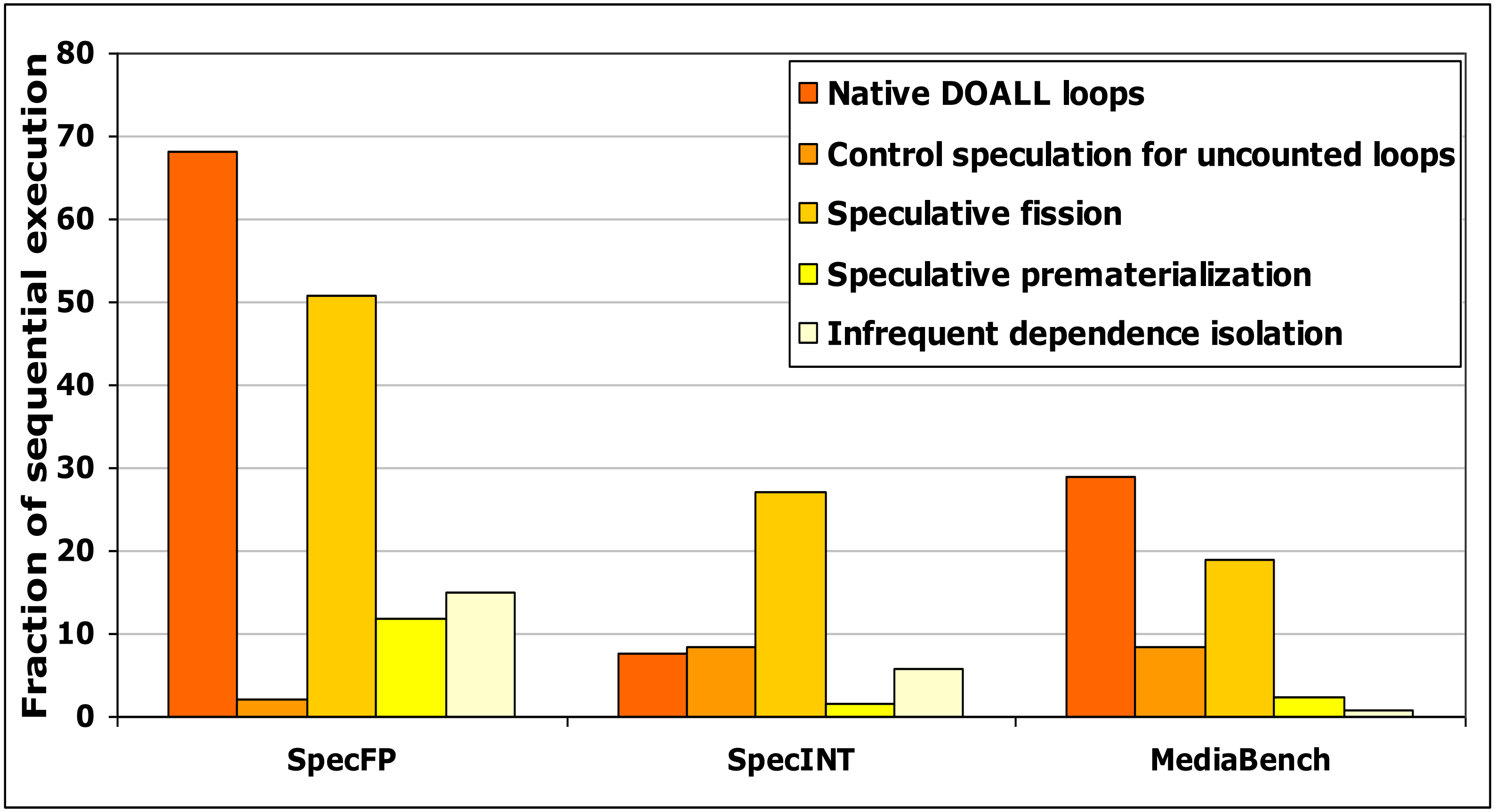

```
   XBEGIN
1':  current =
3':  last =
     for (...) {
1:     current = … ;
2:     work(last);
3:     last = current;
     }
   XCOMMIT
```

- Iterations are sequential due to distance 1 recurrence of the red variables
- Decouple iteration chunks by prematerializing *last* outside the loop

DOALL Coverage – Profiled and Transformed



Coverage Breakdown

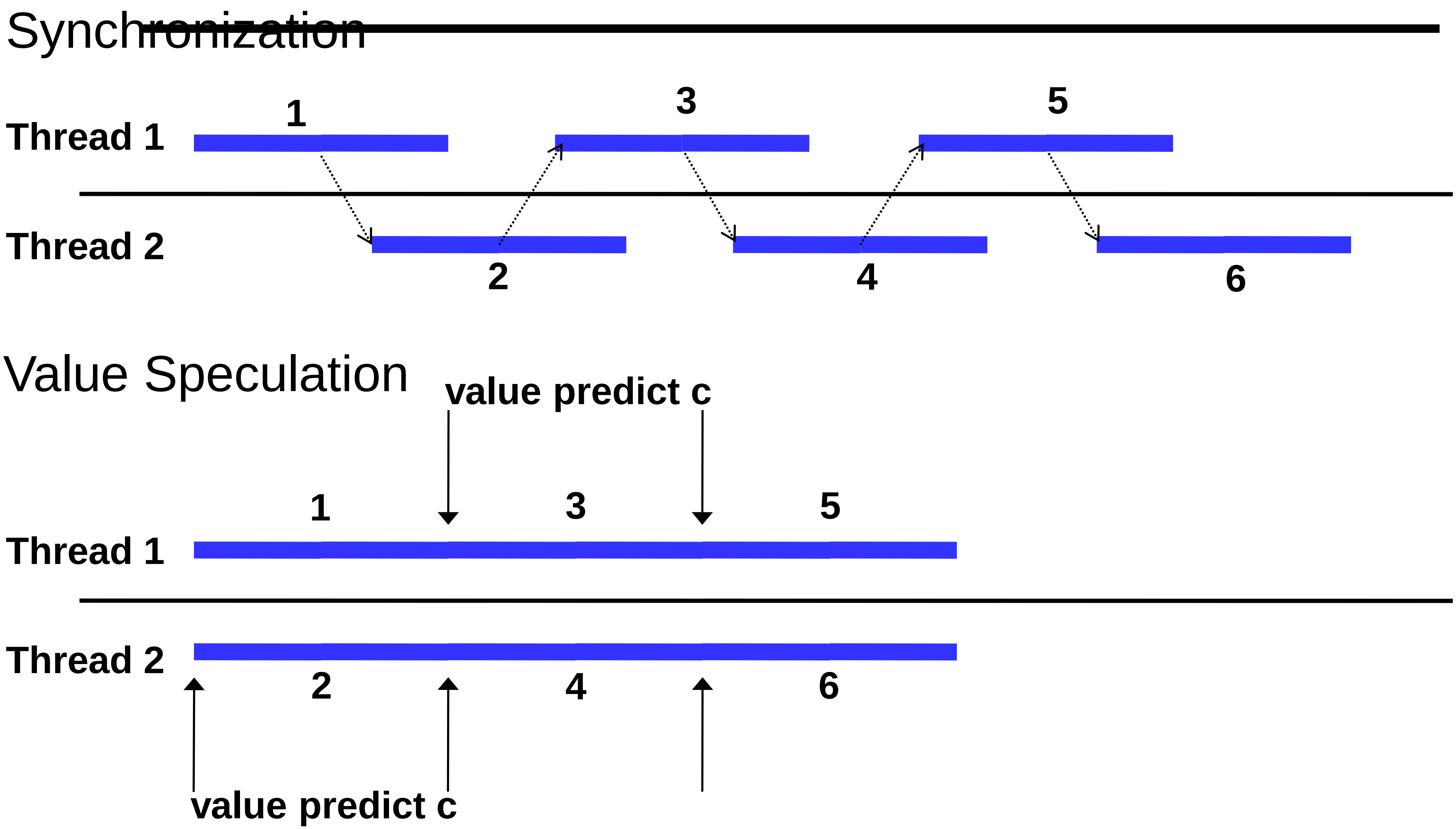


Improving DSWP with Value Speculation Motivating Example

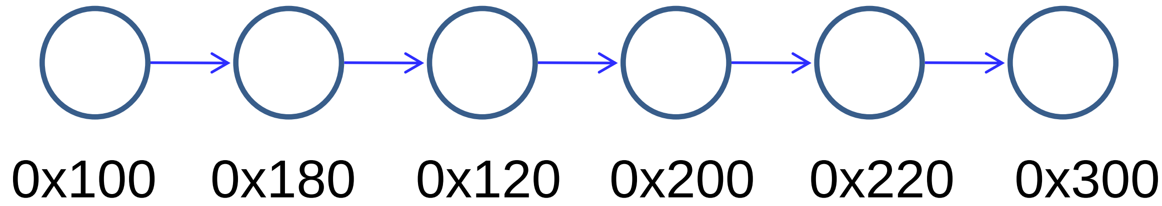
```
l = create_list();
while (...){
   process_list(l);
   modify_list(l);
}
```

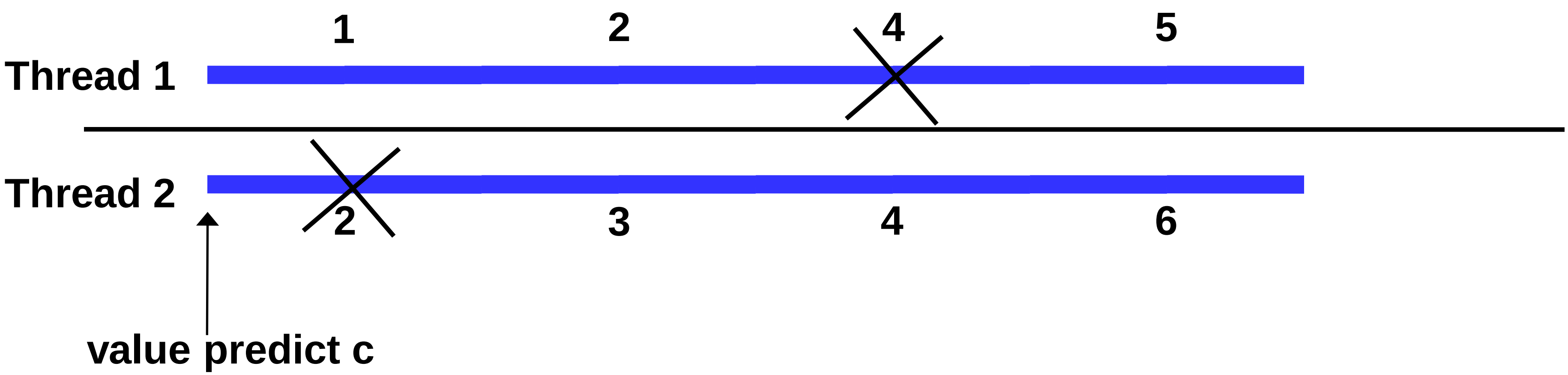
```
while (c != null){
    c1 = c;
    c = c->next_cl;
    int w = c1->pick_weight;
    if (w < wm){
        wm = w;
        cm = c1;
    }
}
```

Need to Break Cross-thread Dependences



Value Speculation



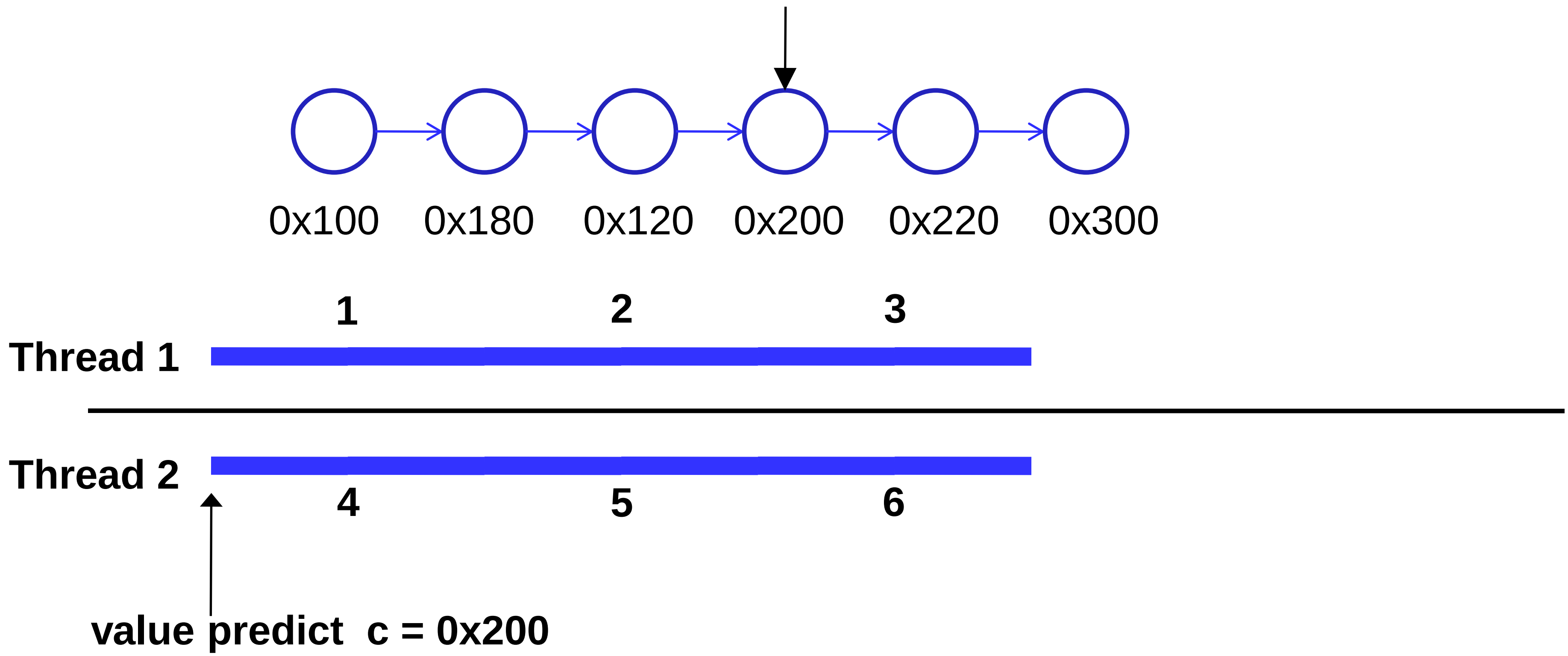


Value Speculation for TLP: Observation 1

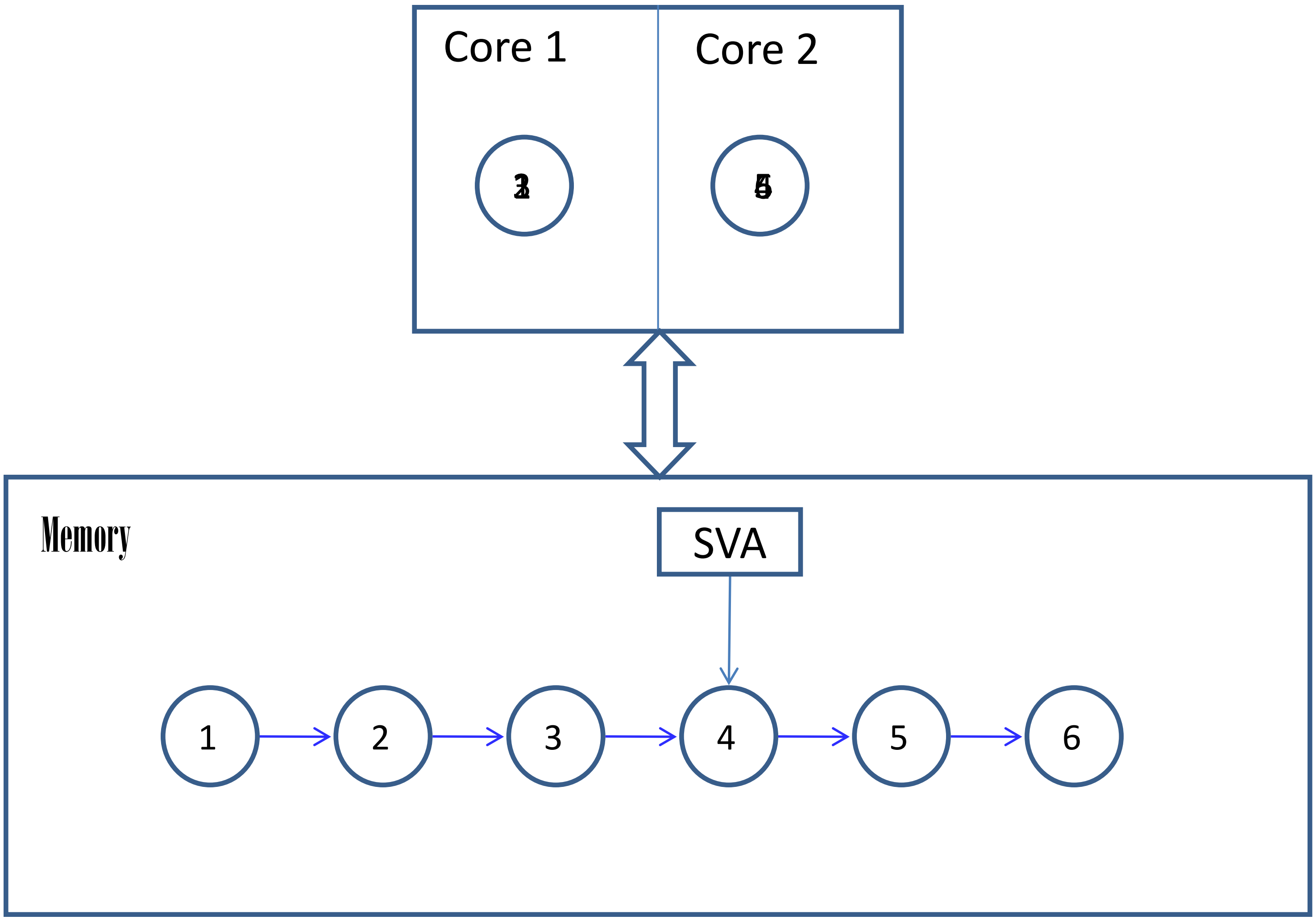
- Predicting a value in a *particular* iteration is harder than predicting *a value* of the operation within a *range* of iterations

Value Speculation for TLP: Observation 2

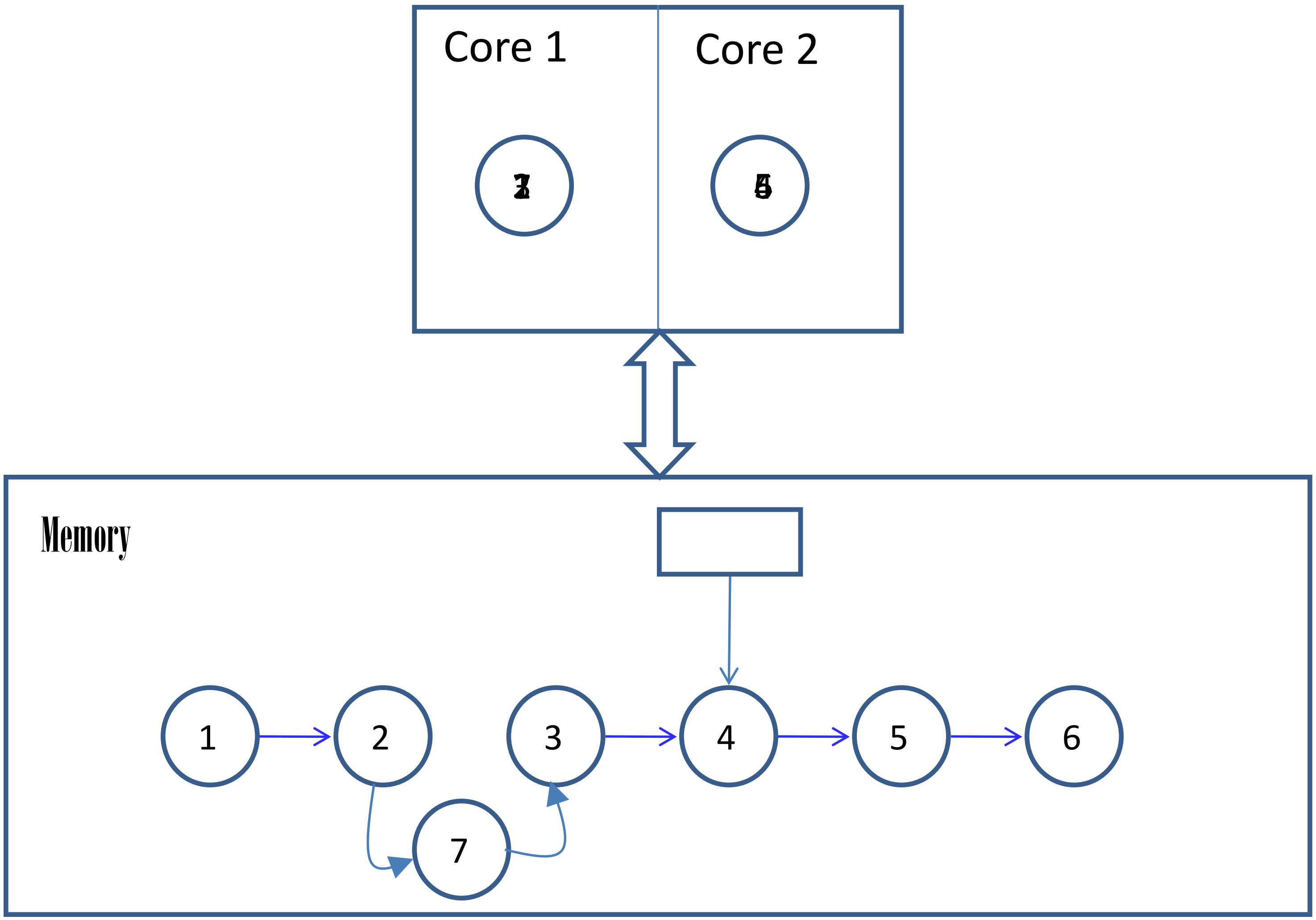
- It is sufficient to predict the values in a few iterations to get TLP



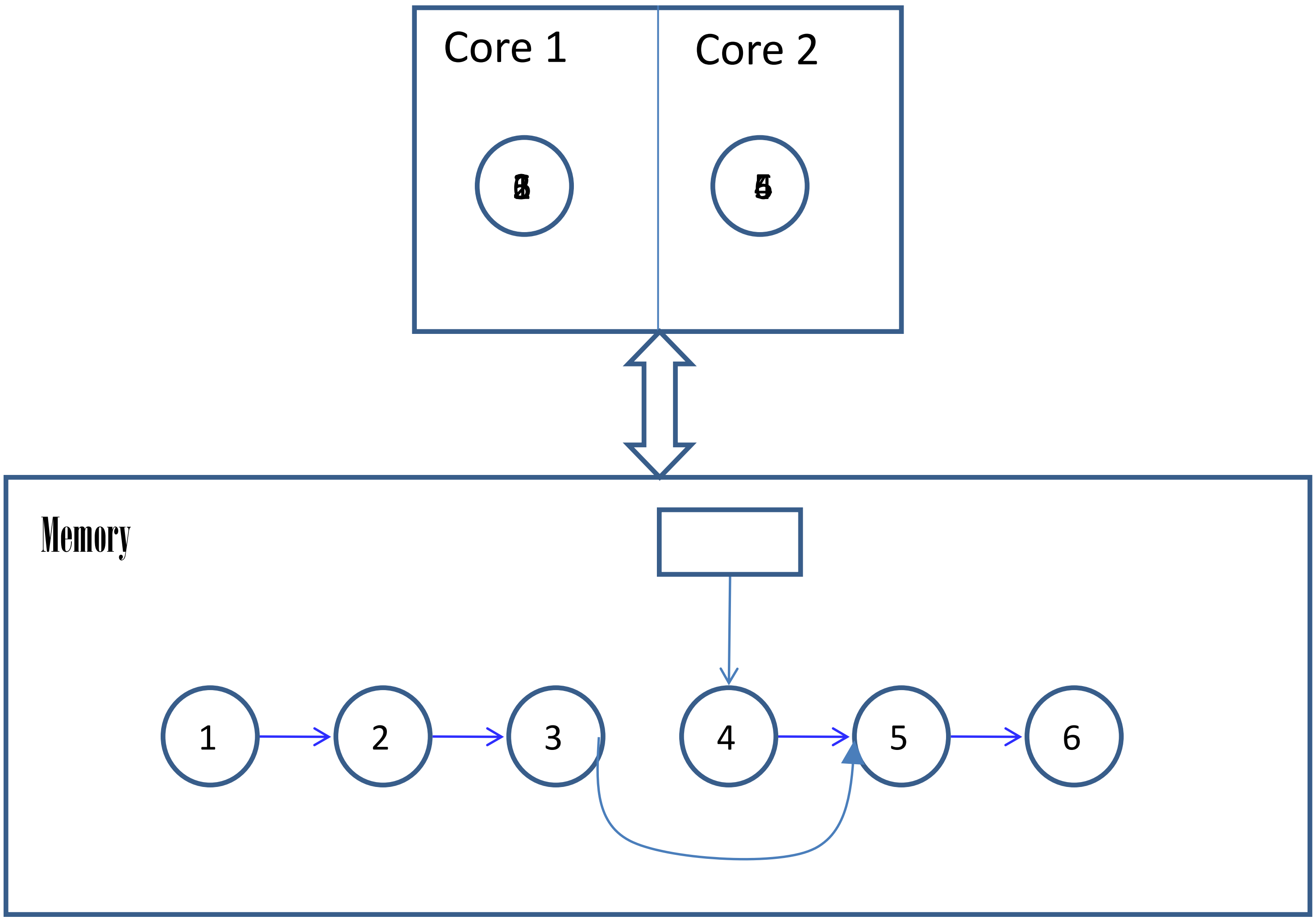
Memoize and Speculative Parallel Execution



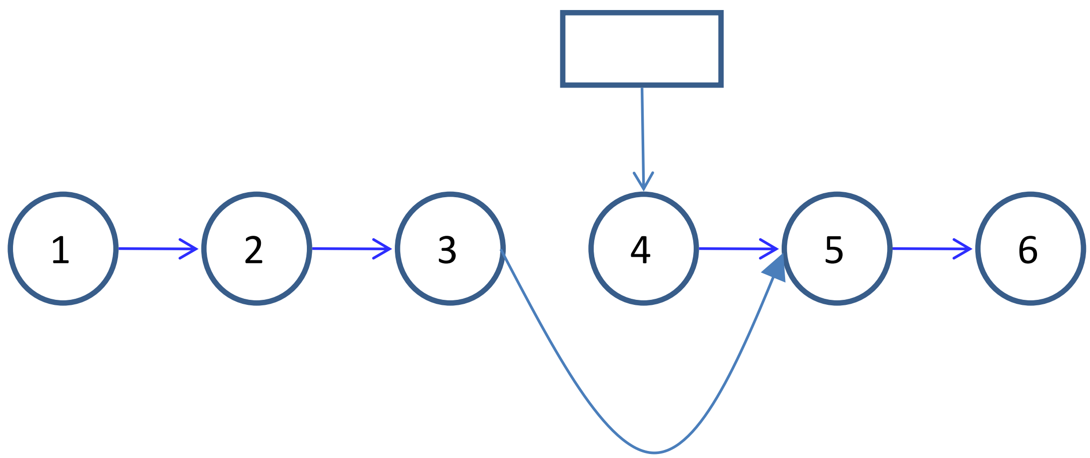
Speculative Parallel Execution



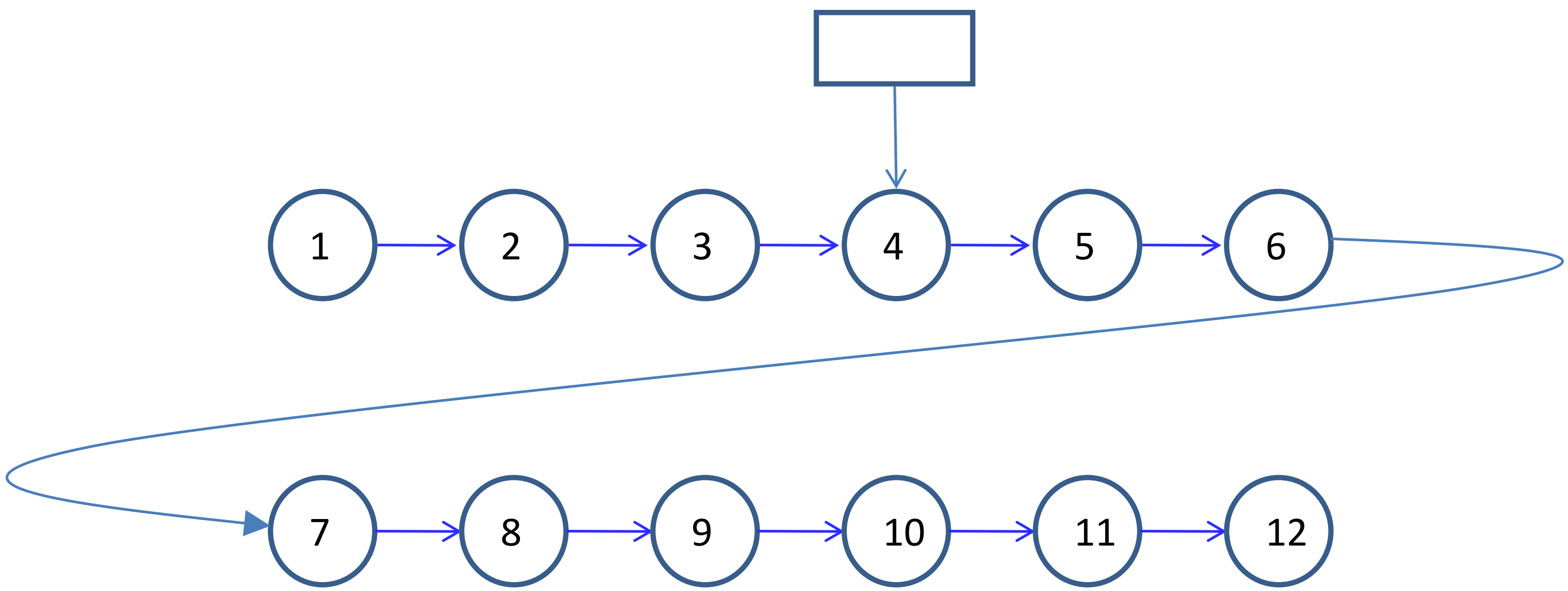
Mis-speculation



Memoizing Once: Problems – Delete One of the Starting Nodes



Memoizing Once: Problems – The List Grows after the Initial Traversal



Discussion Point 1 – Fission vs Value Speculation

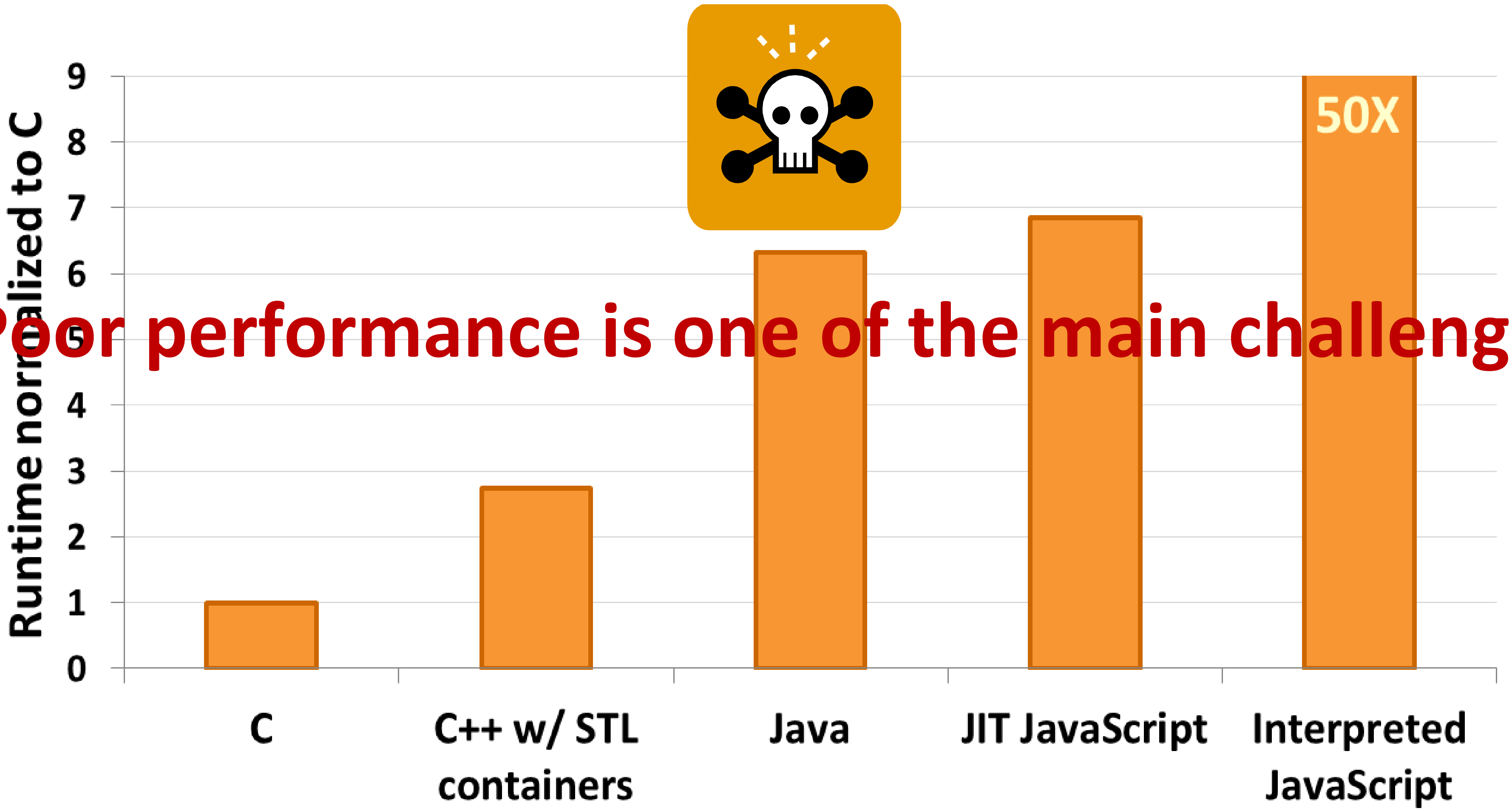
- When is fission better?
- When is value speculation better?
- Is either more powerful than the other?
- Which would you use in your compiler and why?
- Would you be confident your code would run correctly if these transformations were done?

Discussion Point 2 – Dependences

- Are these "tricks" valid for other common data structures?
  - Are any generalizations needed?
- What other common data dependences are we missing in loops?

Client-side Computation in JavaScript

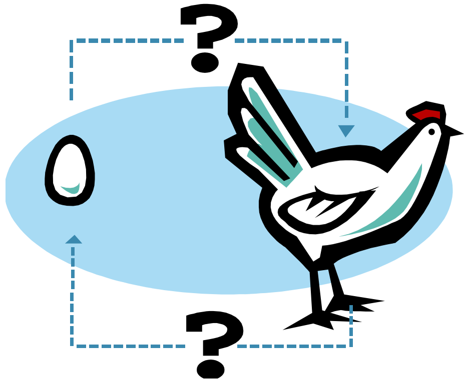
❖ Flexibility, ease of prototyping, and portability



Poor performance is one of the main challenges

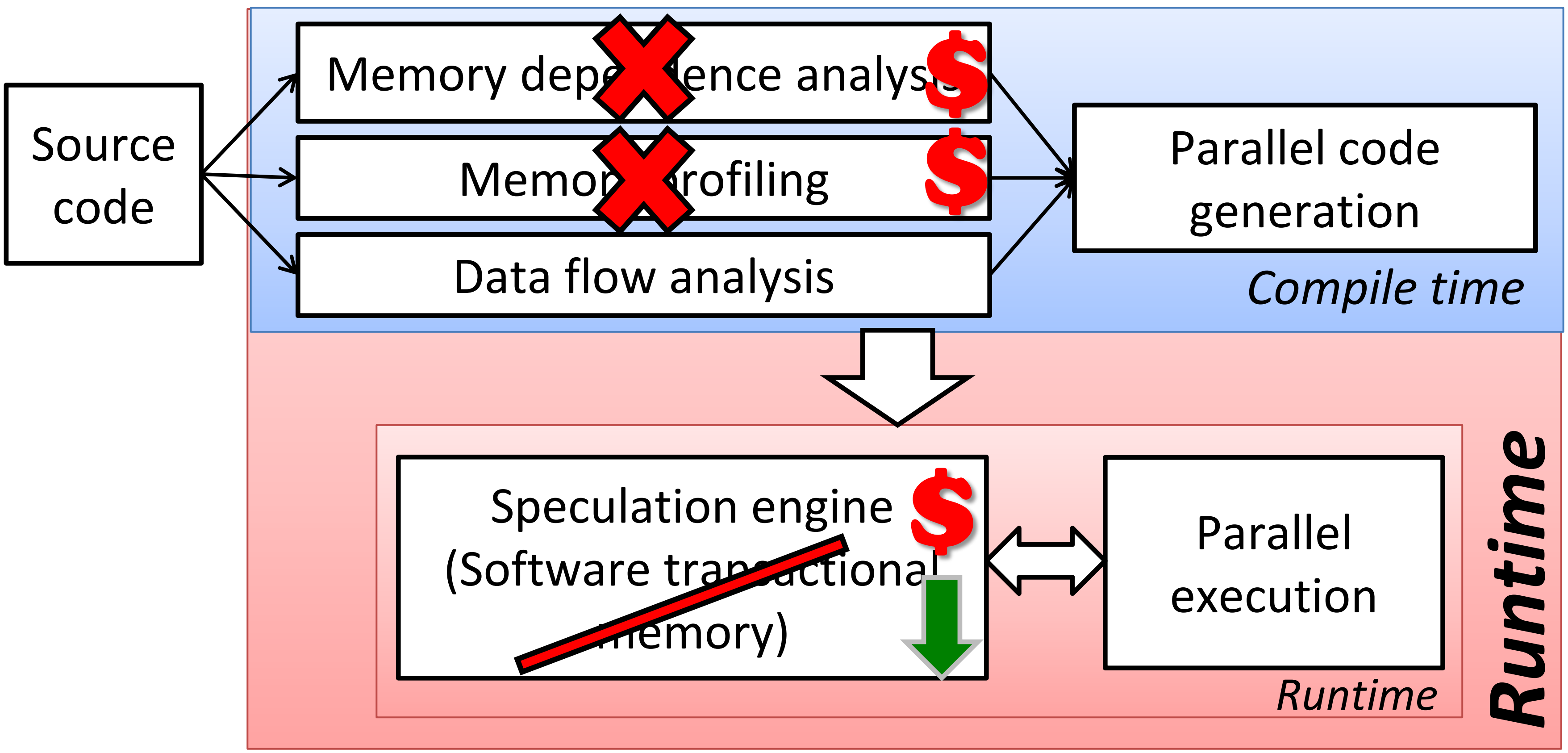
Client-side Applications

- **Interaction-intensive:**
  - Largely composed of event handlers, triggered by user
  - Examples are *Gmail*, *Facebook*, etc.
- **Compute-intensive:**
  - Dominated by loops and hot functions
  - Online image editing such as Adobe's *Photoshop.com,* Google's *Picnik*
  - Lot more potential:
    - Online games
    - Video editing
    - Sound editing and voice recognition



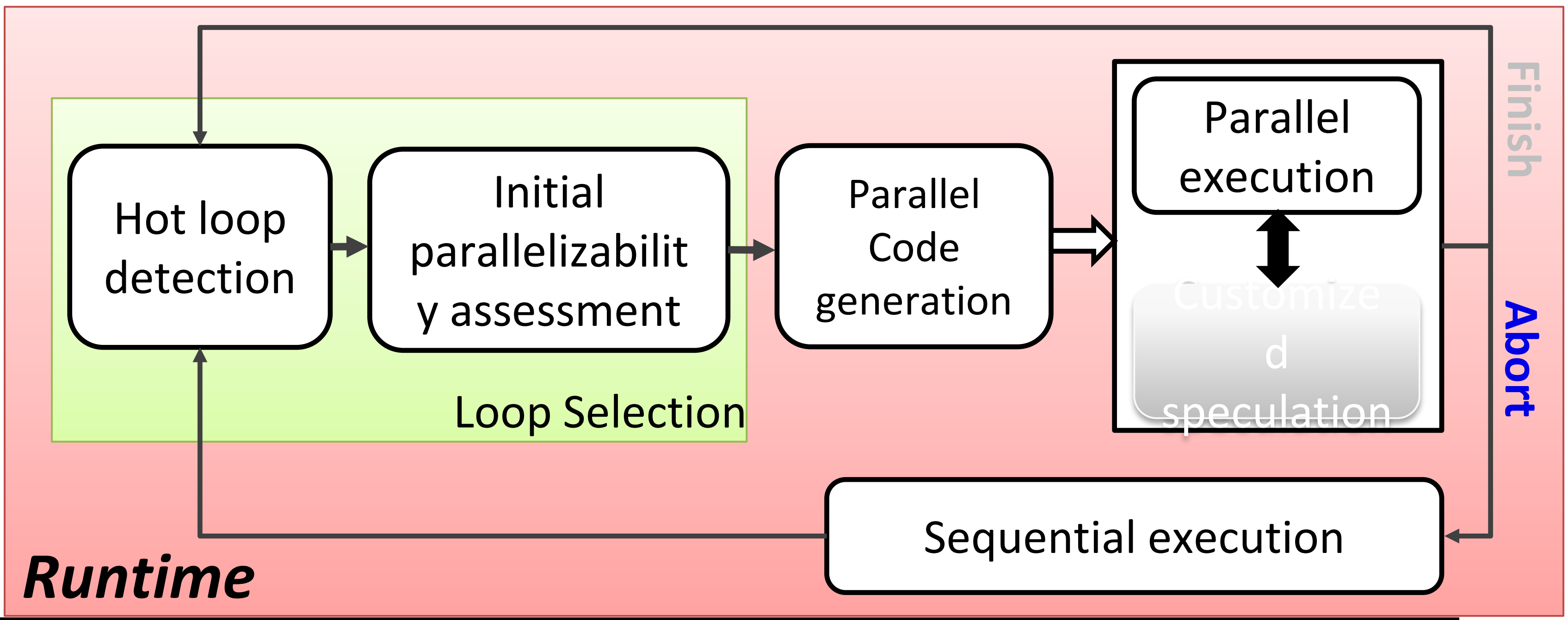
JavaScript Parallelization

- A typical static parallelization flow
- Dynamic parallelization



ParaScript Approach

- Light-weight dynamic analysis & code generation for speculative DOALL loops
- Low-cost customized SW speculation with a single checkpoint

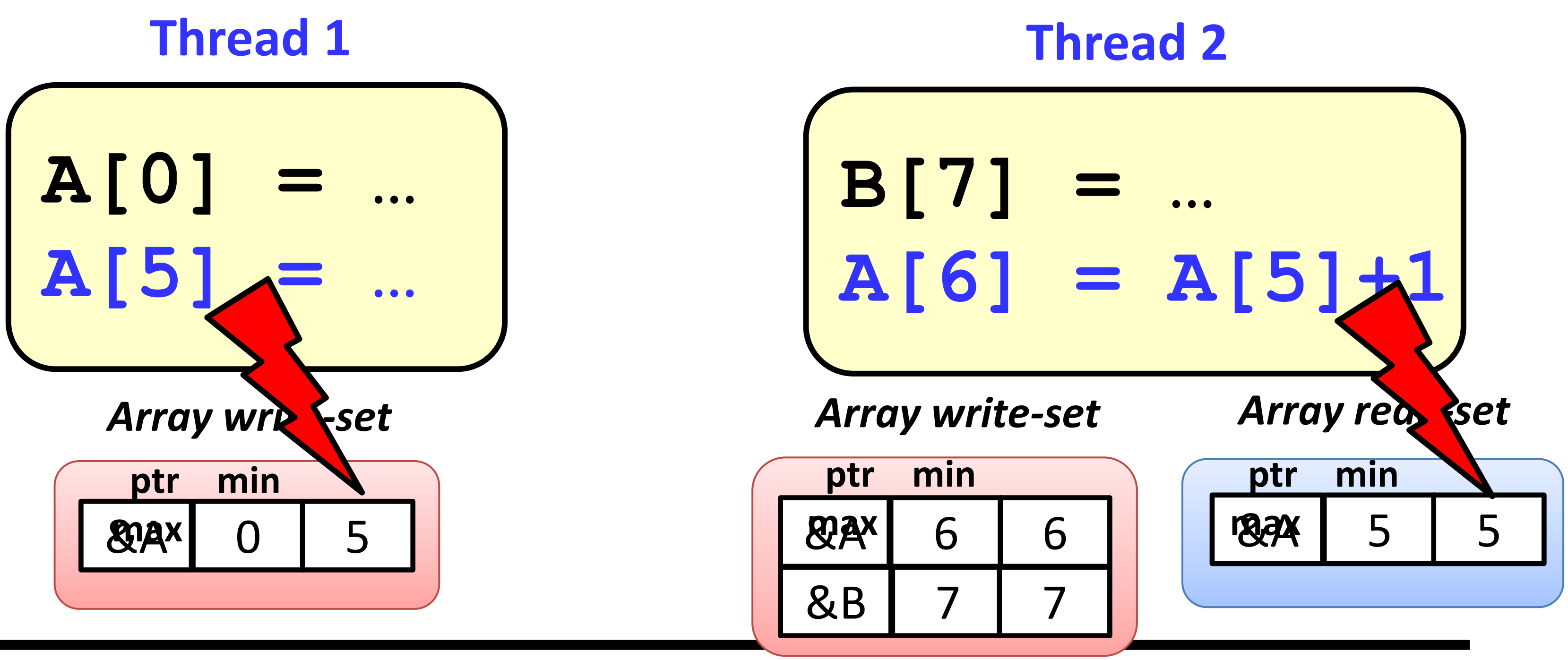


Dependence Analysis

Variable category	Analysis and runtime speculation method
Scalars & Objects	JIT compilation time data flow analysis
Scalar arrays	Runtime initial tests + range-based monitoring
Object arrays	Runtime reference-counting-based monitoring

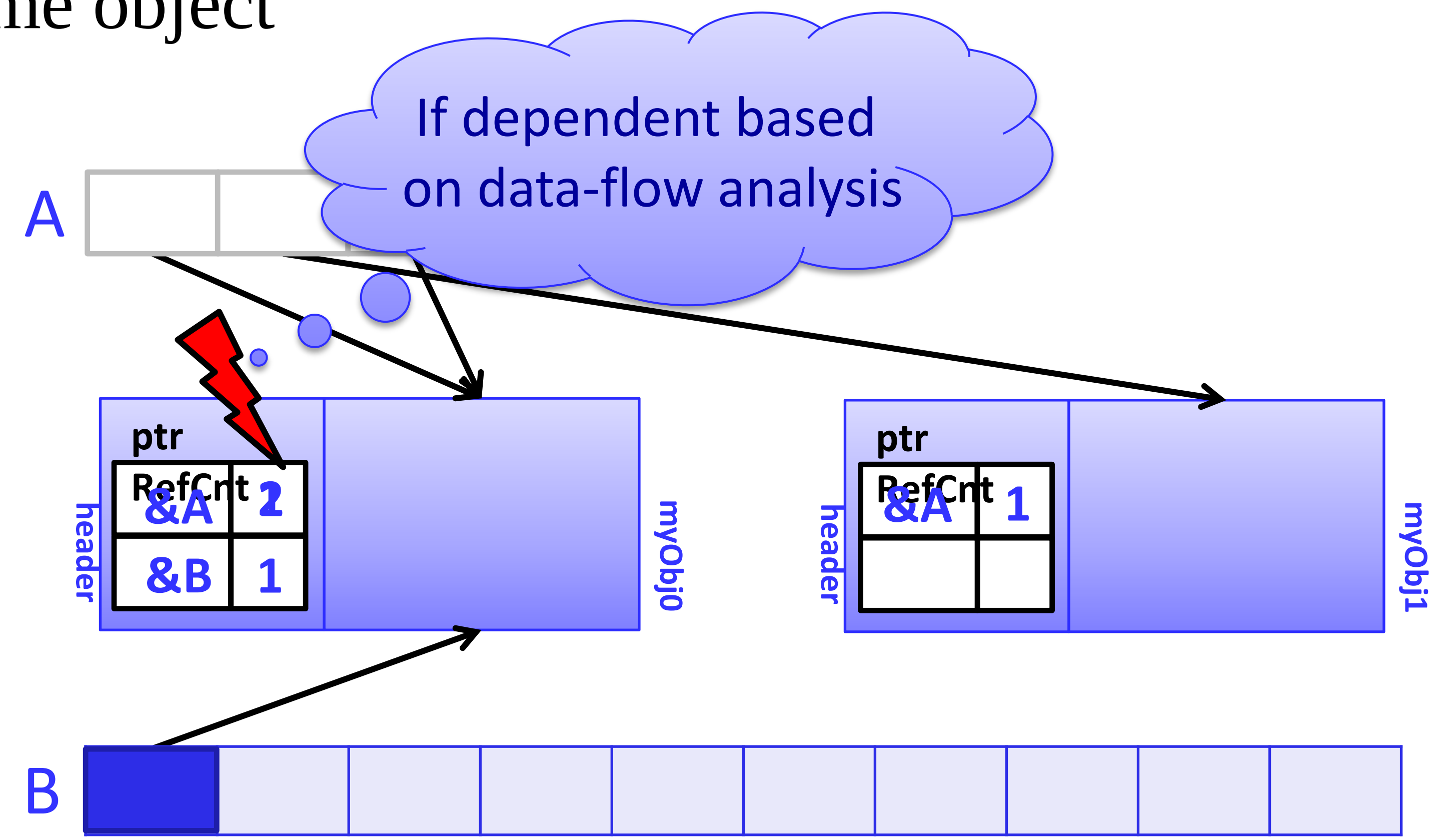
Scalar Array Conflict Detection

- Initial assessment catches trivial conflicts
- Keep track of max and min accessed element indices
- Cross-check RD/WR sets after thread execution



Object Array Conflict Detection

- More involved than scalar arrays
- Different indices of the same array may point to the same object



Loop Selection

- Focus on DOALL-counted (e.g. `for` loops)
- Avoid parallelizing loops with:
  - » Browser interactions
  - » HTTP request functions
  - » Runtime code insertion

Requires locks on browser internals

Requires server-side speculation

Source code	
`var addFunction = new Function("a" , "b", "return a+b;");`	`(a, b){` `}`
`eval("a = 7; b = 13; document.write(a+b);");`	`a = 7; b = 13;` `document.write(a+b);`

Checkpointing Mechanism

- Go through all references, clone them, and ask GC not to touch the clones

Cloning process	
Strings, numbers and Booleans	Copy the values
Custom objects	Deep-copy all properties, avoid recursion
Functions	- Same as custom objects - No need to clone source code
Arrays	Clone all properties and elements

- **Monitor overhead, back out if more expensive than a threshold**

Checkpointing Optimizations

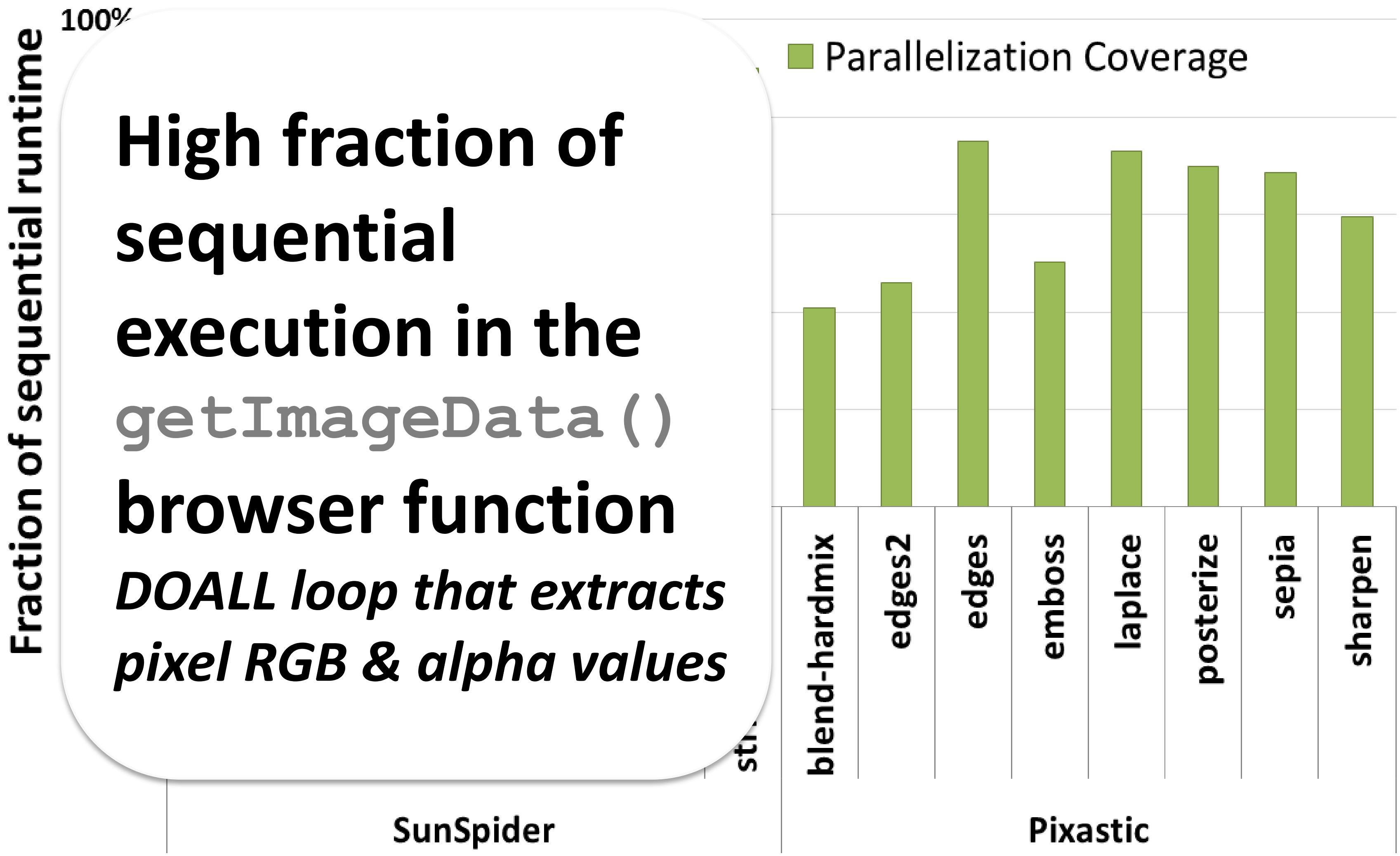
- Selective variable cloning
  - Only clone a variable if it is touched during speculative execution

- Array clone elimination
  - Large arrays holding results of browser functions
  - Instead of cloning the array, just call the function again for recovery
    - E.g. **`getImageData`** in the **`canvas`** HTML5 element

Experimental Setup

- Implemented in Firefox 3.7a1pre
- Subset of SunSpider benchmark suite
  - Others identified as not parallelizable early on, causing 2% slow-down due to the initial analysis.
- A set of Pixastic Image Processing filters
- 8-processor system -- 2 Intel Xeon Quad-cores, running Ubuntu 9.10
- Ran each benchmark 10 times and took the average

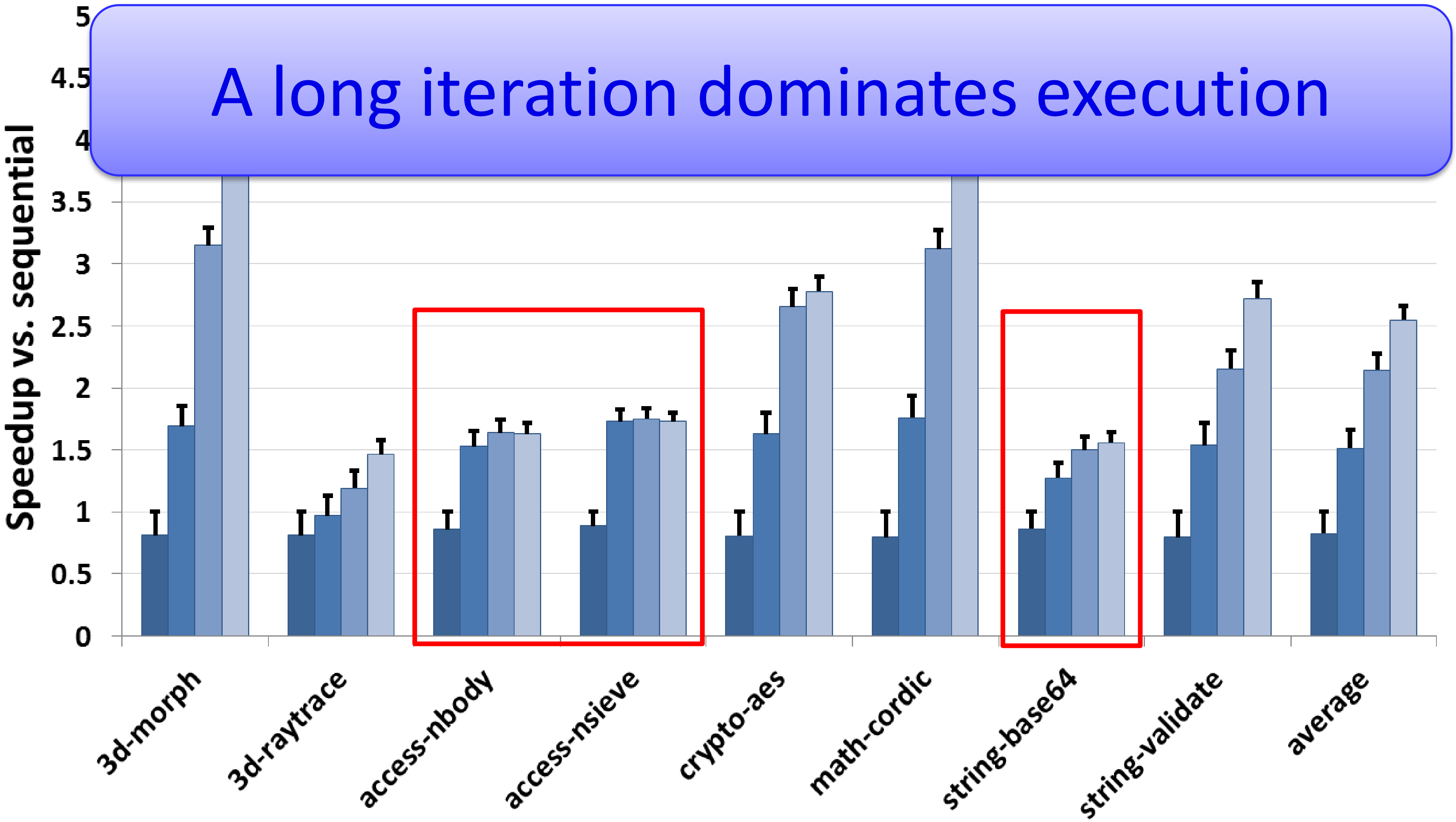
Parallelism Coverage



High fraction of sequential execution in the `getImageData()` browser function

DOALL loop that extracts pixel RGB & alpha values

SunSpider


A long iteration dominates execution
Speedup vs. sequential
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
3d-morph
3d-raytrace
access-nbody
access-nsieve
crypto-aes
math-cordic
string-base64
string-validate
average

Pixastic Image Processing