Colorama: Architectural Support for Data-Centric Synchronization

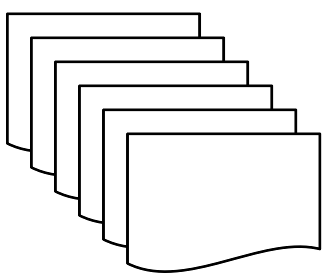
Luis Ceze, Pablo Montesinos, Christoph von Praun, Josep Torrellas

Code-Centric vs. Data-Centric Synchronization

```
declare A,B,C
```

A B C

```
lock L
ld A
st B
st C
unlock L

lock L
st A
st B
unlock L
```



```
declare A,B,C
color red A,B,C
```

A B C

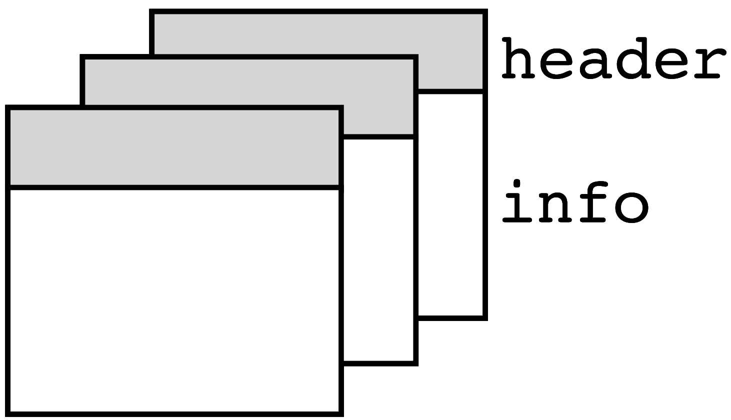
```
ld A
st B
st C

st A
st B
```

inferred critical sections

Code-Centric vs. Data-Centric Synchronization

	CCS	DCS
reasoning	non-local	mostly local
critical sections	explicitly defined	inferred by system
models	Locks, TM	☺

Example from *mysql*



CCS

- declared in a single place
- `header` protected by global lock
  - 29 sites
- `info` protected by its own lock
  - 14 sites

DCS

- `header` fields same color
- each `info` different color

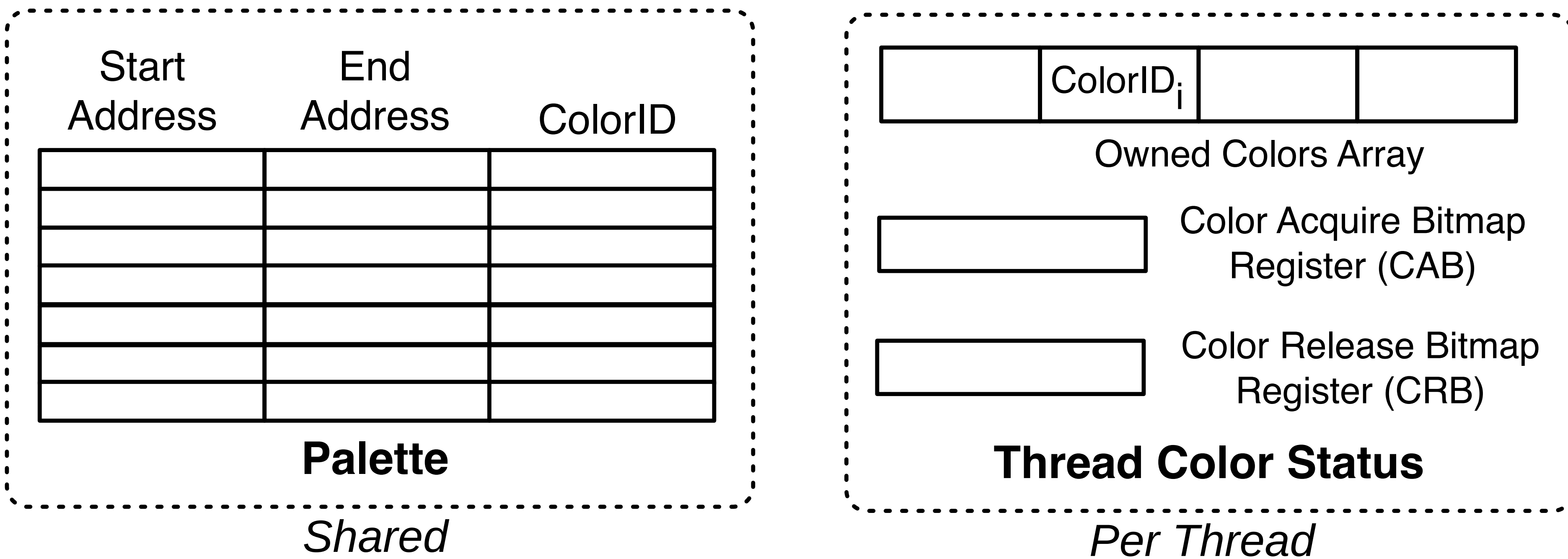
Software DCS

- Software-only DCS concurrently developed [*Vaziri PoPL'06*]
  - for object-oriented languages (Java)
- Needs whole-program analysis
  - might be impractical
- Some code-centric annotations necessary
  - lack of dynamic information
- What about C/C++?

Colorama: Architecture Support for DCS

- Main advantage: cheaply watch all memory references
- Interface to color shared data
- *Enter* critical section if colored data is touched
  - HW checks the color of every memory access
- *Exit* critical section using an exit policy
  - HW provides mechanisms to exit critical sections and enforce policy
- Flexible HW
  - provides the main hooks, software makes decisions

Architecture Components



Colorama Operation Example

```
color A red
color B red
color C red
color E green
color F green
```

Palette

```
A red
B red
C red
E green
F green
```

Thread 1's Owned Colors Array

red
green

thread 1

`ld A` ✓trap to user-level handler, start red critical section

`st B`

`st C`

✓trap to user-level handler, exit red critical section

inferred critical sections

`st E` ✓trap to user-level handler, start green critical section

`st F`

✓trap to user-level handler, exit green critical section

Exiting a Critical Section

- Knowing when to *start* a critical section is easy
- Knowing when to *end* is very hard
- Optimal place undecidable
- Solution is to rely on programming model restrictions
- We use:
    - Return of subroutine where the critical section started

Exit Policy

```
void proc1()
{
  ...
  ...
  <access variable
     with ColorID1>
  ...
  ...
  ...
}
```

ColorID_1 critical section

```
void proc1()
{
  ...
  <access variable
     with ColorID1>
  proc2();
  ...
}
```

ColorID_1 critical section

```
void proc2()
{
  ...
  <access variable
     with ColorID2>
  ...
}
```

ColorID_2 critical section

```
void proc1()
{
  ...
  ...
  <access variable
     with ColorID1>
  ...
  ...
  <access variable
     with ColorID2>
  ...
  ...
}
```

ColorID_2 critical section

ColorID_1 critical section

Intuition Behind Exit Policy

- Functions are natural units of work
- Programmers already think this way
  - empirical data later
- Most bad cases are easily avoided
- Consistent with concurrently developed S-DCS work
  - [Vaziri PoPL'06] uses whole methods as critical sections

Pointer Watching

```
void htUpdate()
{
 ...
 lock(L)
 i = readHash(htPtr)
 ...
 writeHash(htPtr, i)
 unlock(L)
 ...
}
```

Lock-based code

```
color hashTable, red
void htUpdate()
{
 ...
 i = readHash(htPtr)
 ...
 writeHash(htPtr, i)
 ...
}
```

Colorama code

```
color hashTable, red
void htUpdate()
{
 ...
 colorcheck htPtr
 i = readHash(htPtr)
 ...
 colorcheck htPtr
 writeHash(htPtr, i)
 ...
}
```

critical section

Colorama code with colorcheck

- `colorcheck` instructions are inserted by the compiler

Detailed Operation

Color Acquire Bitmap
(CAB register)

Blue	Red	Green

Owned Colors
Array

Blue
Red
Green

f()

```
ld Z
call g()
    stack ← CAB
    CAB ← 0
    ...
    ld A
    ld T
    ld B
    ...
    CRB ← CAB
    CAB ← stack
    ret
```

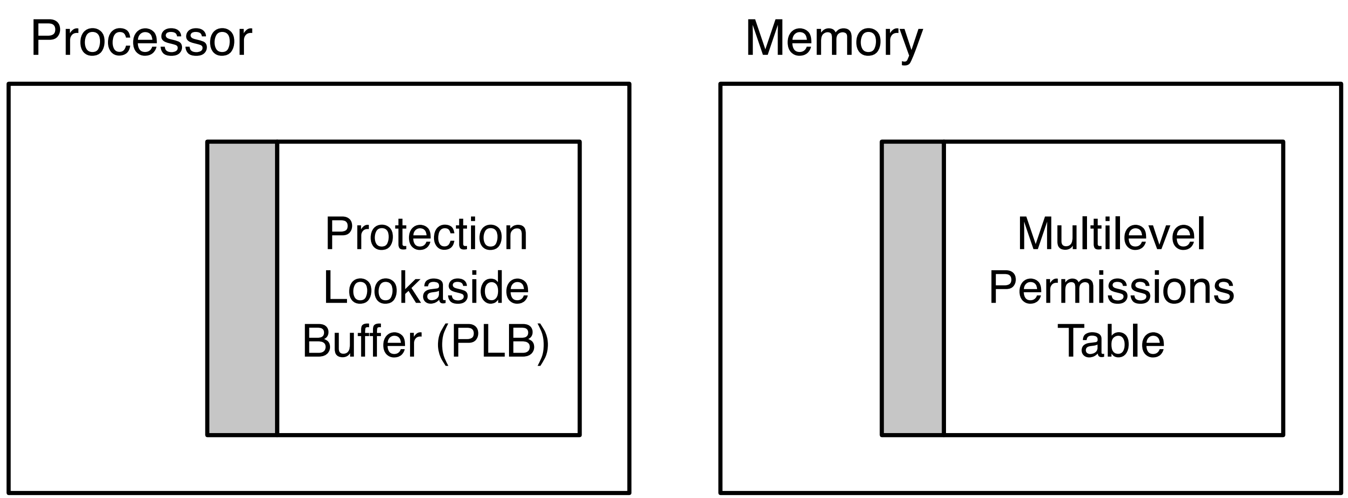
subroutine prologue/epilogue
inserted by the compiler

✓ exits critical sections Red and Green

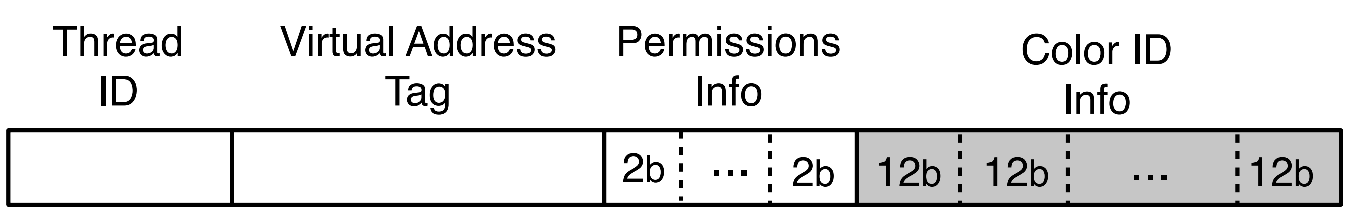
...

Palette Implementation

- Mondrian Memory Protection [Witchel ASPLOS'02]
  - extensions for coloring (shaded)



MMP with the Palette extensions

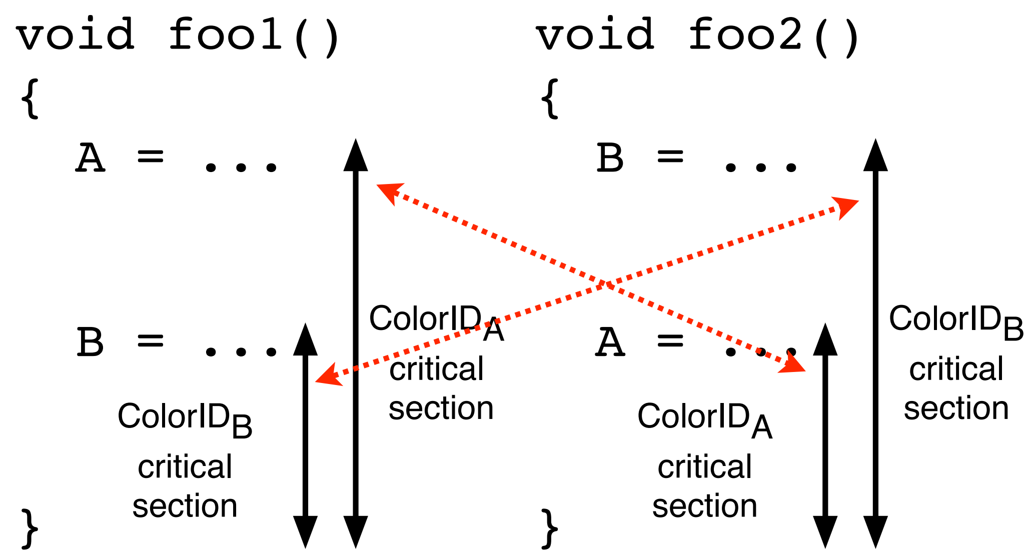


PLB entry

Deadlock Issues in Lock-based Implementation

```
void foo1()
{
  lock(LA)
  A = ...
  unlock(LA)
  lock(LB)
  B = ...
  unlock(LB)
}
```

```
void foo2()
{
  lock(LB)
  B = ...
  unlock(LB)
  lock(LA)
  A = ...
  unlock(LA)
}
```

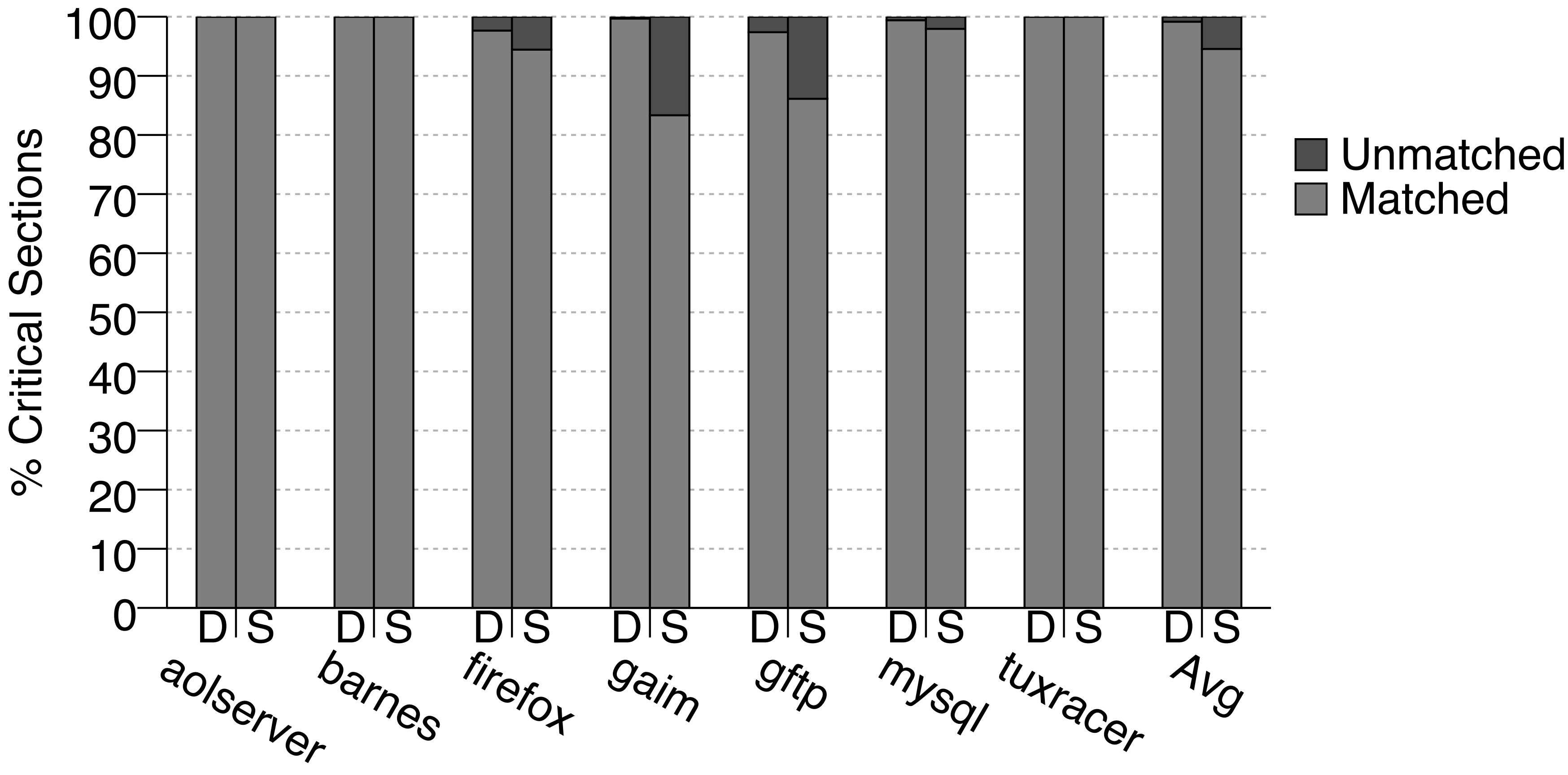


- Inherent limitation of a lock-based Colorama implementation
- TM-based implementation recommended ☺
- Color Ownership Table in memory (SW) for deadlock detection
- Less problems as programmers get used to model

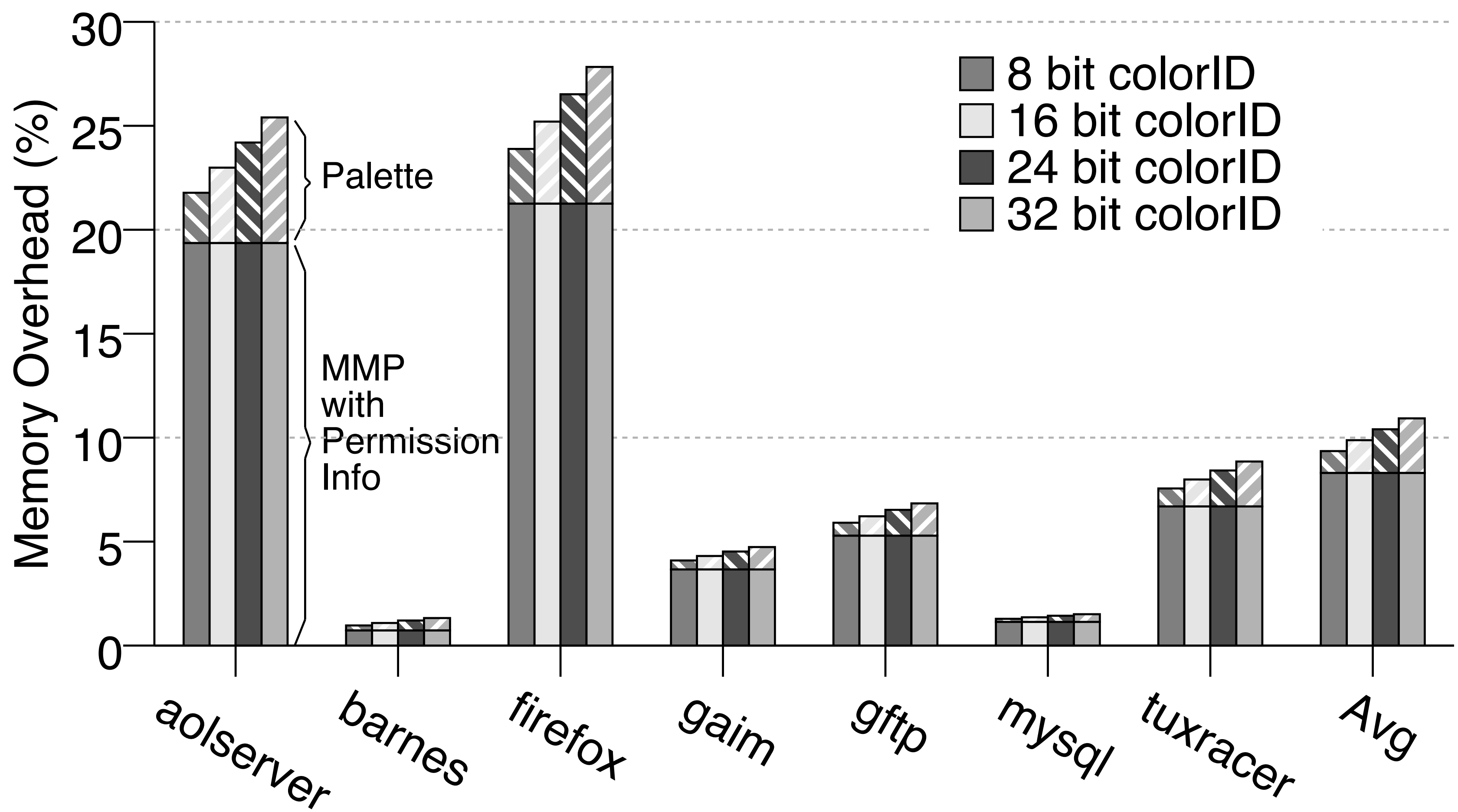
Colorama Evaluation

- No Colorama programs (yet)
- Evaluation consisted in detailed profiling of open-source parallel programs
  - Developed Pin tool to profile critical sections
  - Used MySQL, FireFox, aolserver, tuxracer, ...
- Estimated programming model suitability
- Estimated overheads

Exit Policy Suitability



Memory Overhead of Coloring



Colorama Evaluation Summary

- Programming model apparently suitable
  - few static corner cases, even fewer dynamic
- Overheads tolerable
  - most of the overhead comes from baseline fine-grain memory protection

Colorama Conclusion

- DCS can greatly simplify parallel programming
  - programmer only specifies the colors and follows a simple policy
  - the system, in return, guarantees consistency of shared data
- Hardware has important advantages over a software-only approach

Backup Slides

Code-Centric Synchronization

- Locks and TM are code-centric approaches
  - the programmer explicitly defines code inside the critical sections
- May require *non-local* reasoning
  - changing one critical section implies reasoning about effects on critical sections located in other parts of the program
- Annotations proportional to number of accesses to shared data
- TM is a major simplification over locks
  - can we go beyond that?

Data-Centric Synchronization (DCS)

- Programmer explicitly assigns all shared data to consistency domains
  - typical domains contain multiple data structures
  - domains define sets of data that need to be kept self-consistent
- The system then infers the critical sections automatically
  - guarantees mutual consistency of data inside same domain
- Main benefits: more *local* reasoning
  - programmer thinks about data consistency at declaration time
  - annotations proportional to the number of shared data structures [Vaziri PoPL'06]

CCS x DCS Reasoning

- Non-Local
  - What other parts of the code should I visit to make sure what I did is correct?
  - How do critical sections interfere?
- Local
  - Think about data consistency when creating data-structures
  - the rest should be (mostly) automatic
- Every time shared data is touched
  - programmer needs to insert code for critical sections
  - critical sections exist to keep data consistent, why not annotate data?

Another Example

```
void unrealize (Widget *w)
{
  if (w->realized) {
     lock (L);
     <free structure> /*CRASH*/
     w->realized = false;
     unlock (L);
  }
}
```

Code-centric (with data race)

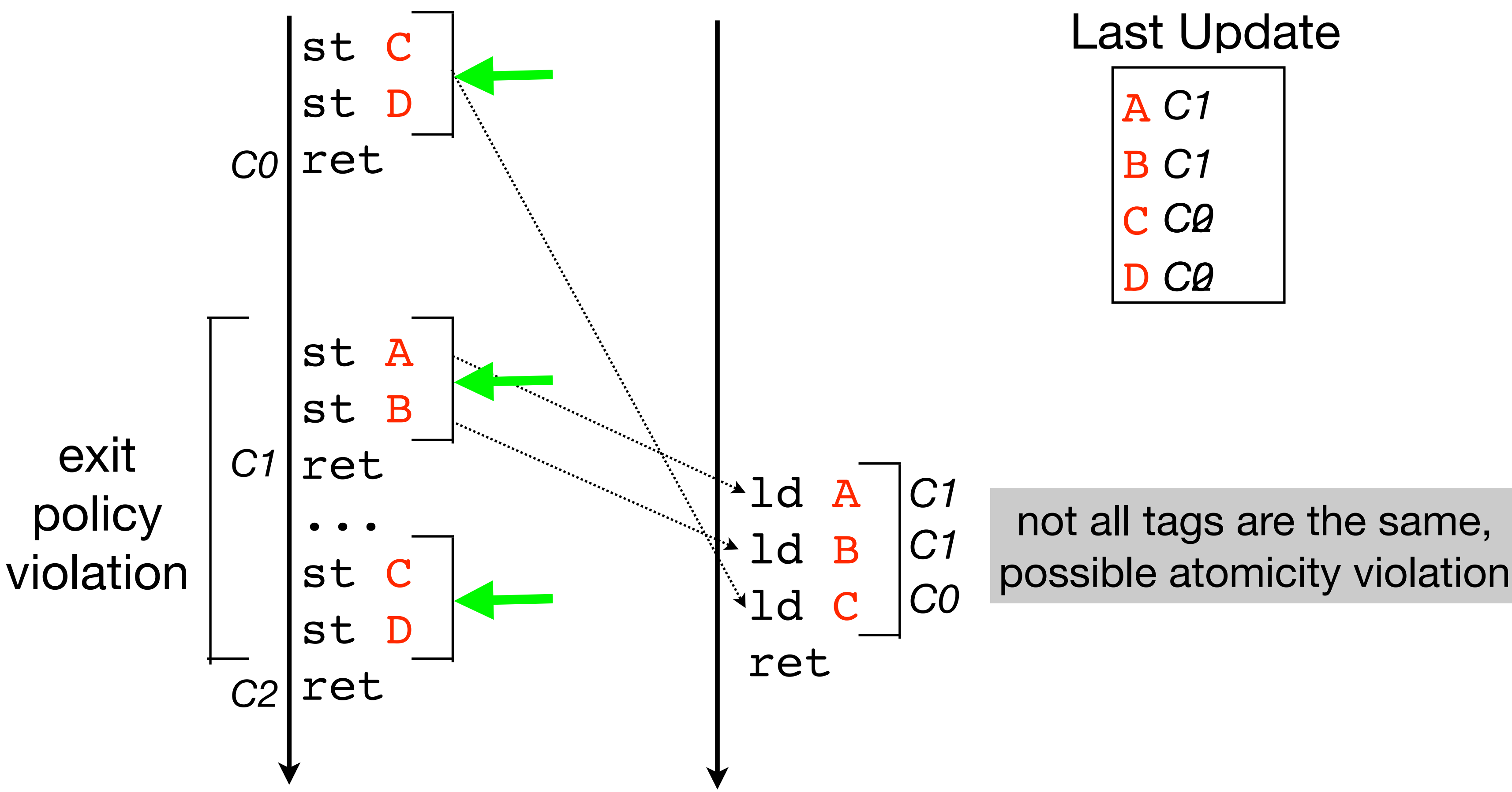
```
color(w, sizeof(*w), GREEN);
void unrealize(Widget *w)
{
  if (w->realized) {
     <free structure>
     w->realized = false;
  }
}
```

Colorama (data-race free)

Monitors

- Conceptually Data-Centric approach to concurrency management
- Programmer still needs to specify what code operates the monitored data (code-centric)
  - monitor interface, needs to be adjusted according to the operation
  - still allow for high-level data-races
- H-DCS is essentially hardware support for very flexible monitors
  - monitor operations are “inferred” from actual code, no need to often redefine monitor interface

Refining Exit Policy - Detecting Partial Updates



Refining Exit Policy - Making It Shorter

```
if T !colored
    CRB ← CAB
st T
CRB ← CAB
st P
st Q
ld R
CRB ← CAB
CAB ← stack
ret
```

possibly local

provably local data (not-colored)

API

System Calls

`color (StartAddr, Size, ColorID)`
`colorprop(StartAddr,Size,ColoredAddr)`
`decolor (Addr)`

Instructions

`colorcheck Addr`
`getcolorid Addr, reg`
`mov reg, CAB`
`mov CAB, reg`
`mov reg, CRB`

Library Calls

`color release ()`
`color release (Addr)`
`color temp release (Addr)`
`color reacquire ()`