Colorama: Architectural Support for Data-Centric Synchronization

Luis Ceze, Pablo Montesinos, Christoph von Praun, Josep Torrellas
Code-Centric vs. Data-Centric Synchronization

**Code-Centric**

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

**Data-Centric**

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

Inferred critical sections:

- `unlock L`
# Code-Centric vs. Data-Centric Synchronization

<table>
<thead>
<tr>
<th></th>
<th>CCS</th>
<th>DCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>reasoning</td>
<td>non-local</td>
<td>mostly local</td>
</tr>
<tr>
<td>critical sections</td>
<td>explicitly defined</td>
<td>inferred by system</td>
</tr>
<tr>
<td>models</td>
<td>Locks, TM</td>
<td>☺</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Start Address</th>
<th>End Address</th>
<th>ColorID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Palette

Shared

Owned Colors Array

Color ID: \( i \)

Color Acquire Bitmap Register (CAB)

Color Release Bitmap Register (CRB)

Thread Color Status

Per Thread
Colorama Operation Example

- **Palette**
  - A red
  - B red
  - C red
  - E green
  - F green

- **Thread 1’s Owned Colors**
  - red
  - green

- **Thread 1**

  - **Inferred critical sections**
    - ld A ✓trap to user-level handler, start red critical section
    - st B
    - st C ✓trap to user-level handler, exit red critical section

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

void proc2()
{
  ...
  ...
  <access variable with ColorID2>
  ...
  ...
  ...
}
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
void htUpdate()
{
...
lock(L)
i = readHash(htPtr)
...
writeHash(htPtr, i)
unlock(L)
...
}

Lock-based code

void htUpdate()
{
...
    i = readHash(htPtr)
...
    writeHash(htPtr, i)
...
}

Colorama code

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

\( f() \)
- \( ld \ Z \)
- \text{call} \ \( g() \)

\begin{align*}
&\text{stack} \leftarrow \text{CAB} \\
&\text{CAB} \leftarrow 0 \\
&\ldots \\
&\text{ld} \ A \\
&\text{ld} \ T \\
&\text{ld} \ B \\
&\ldots \\
&\text{CRB} \leftarrow \text{CAB} \\
&\text{CAB} \leftarrow \text{stack} \\
&\text{ret}
\end{align*}

\textbf{Color Acquire Bitmap (CAB register)}

\begin{itemize}
  \item Blue
  \item Red
  \item Green
\end{itemize}

\textbf{Owned Colors Array}

\begin{itemize}
  \item Blue
  \item Red
  \item Green
\end{itemize}

\( \checkmark \text{exits critical sections Red and Green} \)

\( \checkmark \text{subroutine prologue/epilogue inserted by the compiler} \)
Palette Implementation

• Mondrian Memory Protection [Witchel ASPLOS’02]
  • extensions for coloring (shaded)
Deadlock Issues in Lock-based Implementation

- 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

% Critical Sections

Unmatched
Matched

D S D S D S D S D S D S D S D S D S

aolserver barnes firefox gain gftp mysql tuxracer Avg

Unmatched
Matched
Memory Overhead of Coloring

![Graph showing memory overhead for various applications and color depths.](image-url)
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?
void unrealize (Widget *w) {
  if (w->realized) {
    lock (L);
    <free structure> /*CRASH*/
    w->realized = false;
    unlock (L);
  }
}

Code-centric (with data race)

void unrealize(Widget *w)
{
  if (w->realized) {
     <free structure>
     w->realized = false;
  }
}

Colorama (data-race free)

Another Example

25
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

exit policy violation

Last Update

A C1
B C1
C C0
D C0

not all tags are the same, possible atomicity violation
Refining Exit Policy - Making It Shorter

\[
\begin{align*}
\text{possibly local} & \quad \text{provably local data (not-colored)} \\
\text{if } T \neq \text{colored} & \\
& \quad CRB \leftarrow CAB \\
& \quad st \ T \\
& \quad CRB \leftarrow CAB \\
& \quad st \ P \\
& \quad st \ Q \\
& \quad ld \ R \\
& \quad CRB \leftarrow CAB \\
& \quad CAB \leftarrow \text{stack} \\
& \quad \text{ret}
\end{align*}
\]
### 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 ()`