A Technique for the Effective and Automatic Reuse of Classical Compiler Optimizations on Multithreaded Code

Pramod G. Joisha, Robert S. Schreiber, Prithviraj Banerjee, Hans-J. Boehm, Dhruva R. Chakrabarti

HP Labs, Palo Alto
Impetus for this Work
– Recovering useful knowledge from lowered threaded code
  • Example: Detecting critical sections in a CFG (Control-Flow Graph)

• Are t_2 and t_4 value-equivalent?
  – Assume data-race freedom – race semantics anyway undefined for languages like C and C++
• Absent nested synchronization, t_2 and t_4 are equal
  – Can be established using prior work – e.g., Roach-Motel semantics, Ševčík’s transformations

Source Code

```
0  lock(Q->f);
   ...
1  unlock(Q->f);
```

Intermediate Representation (IR)

```
0' t1 := Q;
1' t2 := t1->f;
2' lock(t2);
   ...
3' t3 := Q;
4' t4 := t3->f;
5' unlock(t4);
```
Problem of Nested Synchronizations

In a data-race-free program, can $t_2$ and $t_4$ be different?

YES!
Nested Nature May Not be Apparent!

```
0  t1 := Q;
1  t2 := t1->f;
2  lock(t2);
3  foo();
4  t3 := Q;
5  t4 := t3->f;
6  unlock(t4);
```

```
foo():
0'  lock(l);
1'  A := 1;
2'  unlock(l);
3'  lock(l);
4'  A := 2;
5'  unlock(l);
6'  return;
```
Motivating Questions

- What’s the effect of cross-thread interactions on $t_2$, $t_4$?
  - How to model and analyze cross-thread interactions?
  - How to automatically utilize the results of such an analysis?
    - We really don’t want to go in and change the existing phases in a compiler
  - We want to reuse existing phases to expose cases of equivalence

- More generally, how to transparently empower sequentially-sound data-flow compiler phases on multithreaded code?
Some Past Work on Parallel-Code Analysis and Optimization

– Devise analyses/optimizations, or modify/extend existing (i.e., sequential) ones
  • May be based on IRs specialized for parallel code
    • E.g., Srinivasan et al. [POPL’93], Knoop & Steffen [TOPLAS’96], Novillo et al. [ICPP’98], Rugina & Rinard [TOPLAS’03]
  • Restrict classical scope to synchronization-free code regions
    • Used in production compilers – e.g., GCC and Open64
    • Presently happens automatically, via the opaque treatment of synchronizations
– Above and other related efforts discussed in paper
Contribution: The SR Technique

- Enables classical analyses and optimizations to be more aggressively applied on multithreaded code
  - “Classical” → sequentially sound data-flow analysis frameworks
    - “Monotone Data Flow Analysis Frameworks”, Kam & Ullman [Acta Informatica’77]

- Optimization scope extended across synchronizations

- No new IR constructs, no modifications to the semantics of existing constructs, no changes to existing phases
  - In fact, existing phases treated as black boxes
SR Technique Overview

Two-step algorithm

- Step 1: Find program references with the Siloed Property
  - Computed using the Procedural Concurrency Graph (PCG)
- Step 2: Reflect them in the IR
  - Remove from the “may-definition” and “may-use” sets of a synchronization

Technique’s motif: “Reusing without Revising”

- Implicitly influence an existing compiler phase by exposing siloed references
Using the Technique in a Compiler

- Injecting the SR phase “energizes” downstream phases
- Works with past, present or future data-flow phases
Digging Deeper

– The Procedural Concurrency Graph (PCG)
  • Informal definition, refinements, demonstration

– Enabling classical optimizations
  • Siloed references: Computation and exposure

– Some results
  • Run-time improvements, compilation times
The Procedural Concurrency Graph (PCG)

Informal definition, refinements, demonstration
What’s a PCG?

– Abstraction that models concurrency and interference

– Undirected graph, plus an **interference function**
  
  • Nodes represent the program’s defined procedures
  • Edges mean “may-happen-in-parallel”
  • Interference function labels edges with interference sets

\[
I_i((f_1, f_2)) = \{l_1, l_2, \ldots, l_k\}
\]

– Used in this work to compute siloed references

– May have other applications too
  
  • Could be used by a code-advisory tool to display concurrency/interference
Refinement-Based Construction

– Start with any conservative solution, then repeatedly refine

– Initial solution: Complete graph with self-loops
  • Initial interference: All references that may conflict
    – Two accesses conflict if they are to the same location and at least one is a write

– Refinements: Eliminate edges, or just prune interference sets
  • Identify non-concurrency and/or non-interference opportunities
    – Refinement 1: Procedures that can only be executed by the main thread
    – Refinement 2: One of the procedures can be executed by a spawned thread
    – Refinement 3: Interference-free opportunity due to thread spawning
    – Refinement 4: Interference-free opportunity due to data-race freedom
Embodiments of Refinements

\[
\text{Refinement 1} \hspace{1cm} E^{j+1} = E^j - \{(a, b) \mid a \notin \mathcal{F}_{\text{SPAWNEE}} \land b \notin \mathcal{F}_{\text{SPAWNEE}}\}
\]

- \(E^j\): PCG edge-set after the \(j\)th application of a refinement
- Refinements use a static classification of procedures
- Formulation details in paper

\[
\text{Refinement 2} \hspace{1cm} E^{j+1} = E^j - \{(a, b) \mid a \notin (\mathcal{F}_{\text{SPAWNEE}} \cup \mathcal{F}_{\text{SPAWNER}} \cup \mathcal{F}_{\text{FOLLOW}})\}
\]
Building the PCG: A Demonstration

Excerpt from SPLASH-2’s FMM benchmark

0 GetArguments();
1 InitGlobalMemory();
2 InitExpTables();
3 CreateDistribution();
4 CREATE(ParallelExecute, ...);
5 WAIT_FOR_END(...);
6 printf(...);
7 PrintTimes();

Refinement 1
\[ E^{j+1} = E^j - \{(a, b) \mid a \notin F_{SPAWEEN} \land b \notin F_{SPAWEEN}\} \]

Refinement 2
\[ E^{j+1} = E^j - \{(a, b) \mid a \notin (F_{SPAWEEN} \cup F_{SPAWEER} \cup F_{FOLLOW})\} \]
Siloed references: Computation and exposure
Read-Siloed and Write-Siloed Properties

- "x is read-siloed on P": When a thread is in P, no other thread writes x
- "x is write-siloed on P": When a thread is in P, no other thread accesses x
- Trait is stronger than accesses to x just being data-race free
A Classification of Statements

- Two classes of statements
  - “Interesting synchronizations”
    - Subset of synchronizations
  - “Unaffected statements”
    - All statements other than interesting synchronizations

\[ \text{may}_{\text{defr}} \text{ may}_{\text{use}} \text{ sets of interesting synchronizations alterable} \]
“Siloed-on-a-Procedure” (SOP) Property

- $z$ is siloed on a procedure $f$ if two conditions are met:
  
  - $z$ is write-siloed on every path $P$ in $f$ in which it may be immediately written at a statement $s$ and is not accessed at any unaffected statement in $\text{stmts}(P)-\{s\}$
  
  - $z$ is read-siloed on every path $P$ in $f$ in which it may be immediately read at a statement $s$ and is not written at any unaffected statement in $\text{stmts}(P)-\{s\}$
Why are SOP References Interesting?

– In general, they can be safely dropped from an interesting synchronization’s $\text{may}_{\text{def}}$ and $\text{may}_{\text{use}}$ sets

– This automatically extends a classical optimization’s scope

– But how do we compute these references?
Computing a Subset of SOP References

It can be shown that this is a subset of SOP references

\[
\bar{I}_i(f) = \bigcup_{f' \in \text{MHP}(f)} I_i((f, f'))
\]

\[
S_i(f) = (R_i(f) \cup W_i(f)) - \bar{I}_i(f)
\]
Some Results

Run-time improvements, compilation times
Experiments Overview

- Implemented the SR phase, in gcc
  - Structured as a loop
    - Each iteration applied the SR Technique once, plus five arbitrarily selected basic optimizations
      - pass_fre, pass_copy_prop, pass_ccp, pass_merge_phi, pass_dce
    - Just prior to SR phase, above optimizations repeatedly applied until IR reaches quiescence

- Produced two executables for each benchmark
  - Baseline: Version with the SR phase turned off
  - Enabled: Version with the SR phase turned on

- Main metric: Relative performance, i.e., enabled/baseline

- Up to 41% performance improvement observed (SPLASH-2)
  - Experiments with 1 to 16 threads
  - Average improvement of 6% across all tested programs over all thread counts
FMM Relative Performance

Execution Time (Seconds)

Relative Performance (Baseline Execution Time/Enabled Execution Time)

Thread Count

Relative Performance
 Baseline Execution Time
 Enabled Execution Time
## Compilation Times (Seconds)

<table>
<thead>
<tr>
<th>Program</th>
<th>Baseline Time ($B$)</th>
<th>SR Technique Statistics</th>
<th>$\frac{A}{B}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total Time</td>
<td>No. of Applications</td>
<td>Average Time ($A$)</td>
</tr>
<tr>
<td>m-fmm</td>
<td>9.36</td>
<td>0.88</td>
<td>3</td>
</tr>
<tr>
<td>ocean-c</td>
<td>26.88</td>
<td>9.95</td>
<td>4</td>
</tr>
<tr>
<td>barnes</td>
<td>6.68</td>
<td>0.58</td>
<td>3</td>
</tr>
<tr>
<td>wr-spl</td>
<td>5.93</td>
<td>0.87</td>
<td>3</td>
</tr>
<tr>
<td>wr-nsq</td>
<td>5.22</td>
<td>2.97</td>
<td>11</td>
</tr>
<tr>
<td>lu-c</td>
<td>2.47</td>
<td>0.36</td>
<td>3</td>
</tr>
<tr>
<td>radix</td>
<td>1.93</td>
<td>0.56</td>
<td>4</td>
</tr>
<tr>
<td>fft</td>
<td>1.75</td>
<td>0.33</td>
<td>3</td>
</tr>
</tbody>
</table>
Conclusions

– SR Technique: Static analysis to reuse legacy static analyses
  • Key idea: Implicitly influence an existing compiler phase using siloed references
  • Undefined data-race semantics (e.g., POSIX Threads, C, C++) can be exploited

– No new IR constructs, no modifications to the semantics of existing constructs, no changes to existing phases
  • Cost-effective way of leveraging legacy compiler portfolios in the multicore era

– Underlying methods will likely have other applications
Questions?