Concurrent SSA Form in the Presence of Mutual Exclusion

Diego Novillo Ron Unrau Jonathan Schaeffer

Department of Computing Science
University of Alberta

1998 International Conference on Parallel Processing
12 Aug 1998

Introduction

- Why explicitly parallel languages?

  ① Automatic parallelization strategies have limited applicability
  ② Popular systems like Java incorporate parallel constructs

- Understanding explicitly parallel languages allows the compiler to

  ① Apply sequential optimizations safely
  ② Introduce new optimizations specific to parallel programs

- We are developing an optimizing compiler framework for explicitly parallel programs

The Problem

- An optimizing compiler for explicitly parallel programs must handle

  ✓ Parallel constructs ✓ Synchronization ✓ Memory conflicts

- Therefore, a sequential compiler may break these programs

If flag is initially 0, constant propagation will create an infinite loop. ⇒

Thread 1	Thread 2
while (flag == 0) ; /* Busy wait */ print(b);	b = compute(); flag = 1;

- Most existing work focuses on correctness issues (race conditions, deadlock detection, programming environments)

- Recent research concentrates on optimization issues (but different synchronization constructs)

Goals and Contributions

1. Develop a framework to analyze and optimize explicitly parallel programs

   ☞ We introduce the CSSAME form → An SSA framework for EPPs with mutual exclusion synchronization

2. Adapt sequential optimization techniques

   ☞ We show how CSSAME can improve concurrent constant propagation without modifications to the original algorithm

   ☞ We adapt a sequential dead-code elimination algorithm

3. Develop new optimization techniques that take advantage of parallel and synchronization structure

   ☞ We introduce Lock Independent Code Motion → A new optimization to reduce size of critical sections

Language Model

- Parallel threads share same address space with interleaving semantics
- Parallelism specified with `cobegin/coend` (for now)
- Synchronization is explicit
  ① Mutual exclusion → `lock/unlock`
  ② Event variables → `set/wait`
  ③ Thread join points → `coend`

```
flag = 0;
cobegin
    T 1: begin
        while (flag == 0)
            ;  /* Busy wait */
        print(b);
    end

    T 2: begin
        b = compute();
        flag = 1;
    end
coend
```

CSSA Form [Lee, Midkiff and Padua]

Original program	CSSA Form
	cobegin
	T 1: **begin**
	lock(L_0);
cobegin	$a_1 = 5$;
T 1: **begin**	$a_3 = \pi(a_1, a_2)$;
lock(L);	$b_1 = a_3 + 3$;
a = 5;	$a_4 = \pi(a_1, a_2)$;
b = a + 3;	$x_1 = b_1 * a_4$;
x = b ∗ a;	unlock(L_0);
unlock(L);	**end**
end	
	T 2: **begin**
T 2: **begin**	lock(L_0);
lock(L);	$b_2 = \pi(b_0, b_1)$;
a = b + 6;	$a_2 = b_2 + 6$;
unlock(L);	unlock(L_0);
end	**end**
coend	**coend**
print(x, a);	$a_5 = \phi(a_1, a_2)$;
	print(x_1, a_5);

The CSSAME Form I

- Refines the CSSA form by reducing number of memory conflicts

  ① CSSA only recognizes `set`/`wait`
  ② CSSAME adds support for `lock`/`unlock`

- Key observation

  Mutual exclusion sections serialize execution $\Rightarrow$ some memory conflicts between them might disappear

- When are memory conflicts superfluous?

  ① **Successive kills** $\rightarrow$ Only last def is exposed out of mutex body
  ② **Protected uses** $\rightarrow$ First def inside mutex body hides conflicts

The CSSAME Form II

① Consecutive kills

```
cobegin
    T 1: begin
        lock(L0);
        a1 = ...
        . . .
        a2 = ...
        unlock(L0);
    end

    T 2: begin
        lock(L0);
        . . .
        a3 = π(a0, a1, a2);
        -------------------
        = a3;
        unlock(L0);
    end
coend
```

② Protected uses

```
cobegin
    T 1: begin
        lock(L0);
        . . .
        a1 = ...
        a3 = π(a1, a2);
        ---------------
        = a3;
        unlock(L0);
    end

    T 2: begin
        lock(L0);
        . . .
        a2 = ...
        unlock(L0);
    end
coend
```

Computing the CSSAME Form

1. Build flow graph for the program

2. Identify mutex structures

3. Compute CSSA form

   ① Get partial ordering between conflicting statements
   ② Place ϕ-terms (standard SSA algorithm)
   ③ Place π-terms

4. Rewrite π-terms

   ① Eliminate arguments that comply with mutex body properties
   ② π-terms with one argument left can be safely removed

Optimizations I – Constant Propagation

CSSA Form

cobegin
  T 1: **begin**
    lock(L_0);
    $a_1 = 5$;
    $a_3 = \pi(a_1, a_2)$;
    $b_1 = a_3 + 3$;
    $a_4 = \pi(a_1, a_2)$;
    $x_1 = b_1 * a_4$;
    unlock(L_0);
  end

  T 2: **begin**
    lock(L_0);
    $b_2 = \pi(b_0, b_1)$;
    $a_2 = b_2 + 6$;
    unlock(L_0);
  end
coend
$a_5 = \phi(a_1, a_2)$;
print(x_1, a_5);

CSSAME Form

cobegin
  T 1: **begin**
    lock(L_0);
    $a_1 = 5$;
    $b_1 = a_1 + 3$;
    $x_1 = b_1 * a_1$;
    unlock(L_0);
  end

  T 2: **begin**
    lock(L_0);
    $b_2 = \pi(b_0, b_1)$;
    $a_2 = b_2 + 6$;
    unlock(L_0);
  end
coend
$a_3 = \phi(a_1, a_2)$;
print(x_1, a_3);

Constant Propagation

cobegin
  T 1: **begin**
    lock(L_0);
    $a_1 = 5$;
    $b_1 = 8$;
    $x_1 = 40$;
    unlock(L_0);
  end

  T 2: **begin**
    lock(L_0);
    $b_2 = \pi(b_0, b_1)$;
    $a_2 = b_2 + 6$;
    unlock(L_0);
  end
coend
$a_3 = \phi(a_1, a_2)$;
print(x_1, a_3);

Optimizations II – Dead Code Elimination

<table>
<tr><th>CSSA Form</th><th>CSSAME Form</th><th>Dead Code Elimination</th></tr>
<tr>
<td>

cobegin
T 1: **begin**
lock(L_0);
$a_1 = foo_0$;
$b_1 = 8$;
$a_2 = b_1 * foo_0$;
unlock(L_0);
end

T 2: **begin**
lock(L_0);
$a_3 = \pi(a_0, a_1, a_2)$;
$b_2 = a_3 + 6$;
unlock(L_0);
end
coend
$b_3 = \phi(b_1, b_2)$;
print(a_2, b_3);

</td>
<td>

cobegin
T 1: **begin**
lock(L_0);
$a_1 = foo_0$;
$b_1 = 8$;
$a_2 = b_1 * foo_0$;
unlock(L_0);
end

T 2: **begin**
lock(L_0);
$a_3 = \pi(a_0, a_2)$;
$b_2 = a_3 + 6$;
unlock(L_0);
end
coend
$b_3 = \phi(b_1, b_2)$;
print(a_2, b_3);

</td>
<td>

cobegin
T 1: **begin**
lock(L_0);
$b_1 = 8$;
$a_2 = b_1 * foo_0$;
unlock(L_0);
end

T 2: **begin**
lock(L_0);
$a_3 = \pi(a_0, a_2)$;
$b_2 = a_3 + 6$;
unlock(L_0);
end
coend
$b_3 = \phi(b_1, b_2)$;
print(a_2, b_3);

</td>
</tr>
</table>

Optimizations III – Lock Independent Code Motion

- A statement is **lock independent** if it references non-conflicting variables
- The algorithm hoists lock independent statements out of the mutex body

```
cobegin
  T 1: begin
    lock(L0);
    b1 = 8;
  ↑ x1 = foo0;
    unlock(L0);
  end

  T 2: begin
    lock(L0);
    b2 = π(b0, b1);
    a1 = b2 + 6;
    unlock(L0);
  end
coend
print(x1);
```

```
cobegin
  T 1: begin
    x1 = foo0;
    lock(L0);
    b1 = 8;
    unlock(L0);
  end

  T 2: begin
    lock(L0);
    b2 = π(b0, b1);
    a1 = b2 + 6;
    unlock(L0);
  end
coend
print(x1);
```

Current and Future Work

- **Current work**

  ① Implemented in SUIF
  ② New optimization techniques: single-writer/multiple-readers, code sinking, lock picking, lock partitioning, partial lock independence
  ③ Support for SPMD parallelism → barriers are another form of mutual exclusion
  ④ Applying techniques to Java

- **Future work**

  ① Apply IPA to propagate mutual exclusion information
  ② Adapt other scalar optimizations
  ③ Cost/benefit analysis. Can we use the same models used in scalar optimizations?