Load Elimination in the Presence of Side-Effects, Concurrency and Precise Exceptions

Christoph von Praun
Florian Schneider and
Thomas R. Gross

Laboratory for Software Technology
ETH Zurich,
Zurich, Switzerland

Motivation

- Frequent occurrence of path-expressions in OO programs:

```
l1 = o.f1.f2

...

l2 = o.f1.f2
```

```
t1 = ld(o, f1);
t2 = ld(t1, f2);
l1  = t2;
...
t3 = ld(o, f1);
t4 = ld(t3, f2);
l2  = t4;
```

- Large number of (indirect) memory accesses
- Irregular access patterns (pointer-chasing)

Load elimination

Goal: Reduce # of memory accesses

"Promote" heap to local vars / registers

```
t1 = ld(o, f1);
t2 = ld(t1, f2);
l1  = t2;
...
t3 = ld(o, f1);
t4 = ld(t3, f2);
l2 = t4;
```

→

```
t1 = ld(o, f1);
t2 = ld(t1, f2);
l1 = t2;
...
l2 = t2;
```

Implementation for Java must consider …

- Control- and data-flow
- Side-effects at call sites
- Precise exceptions
- Multi-threading

Multi-threading (1/3)

Original program:

```
s1,s2 = 0;      // shared
l1,l2,l3 = 0; // local to thread1

// thread 1                  // thread 2
l1 = ld(s1);                 st(s1, 1);
l2 = ld(s2);                 st(s2, 1);
if (l2 != 0) {
   l3 = ld(s1);
}
```

Possible results : `(l2,l3)`

- SC: {`(0,0),(1,1)`}
- JC : {`(0,0),(1,1),(1,0)`}

Subset correctness [Lee et. al. PPoPP 99]:

- Results of optimized programs must be in that set.

Multi-threading (2/3)

Optimized (load-elimination):

```
// thread 1              // thread 2
l1 = ld(s1);             st(s1, 1);
l2 = ld(s2);             st(s2, 1);
if (l2 != 0) {
    l3 = l1;
}
```

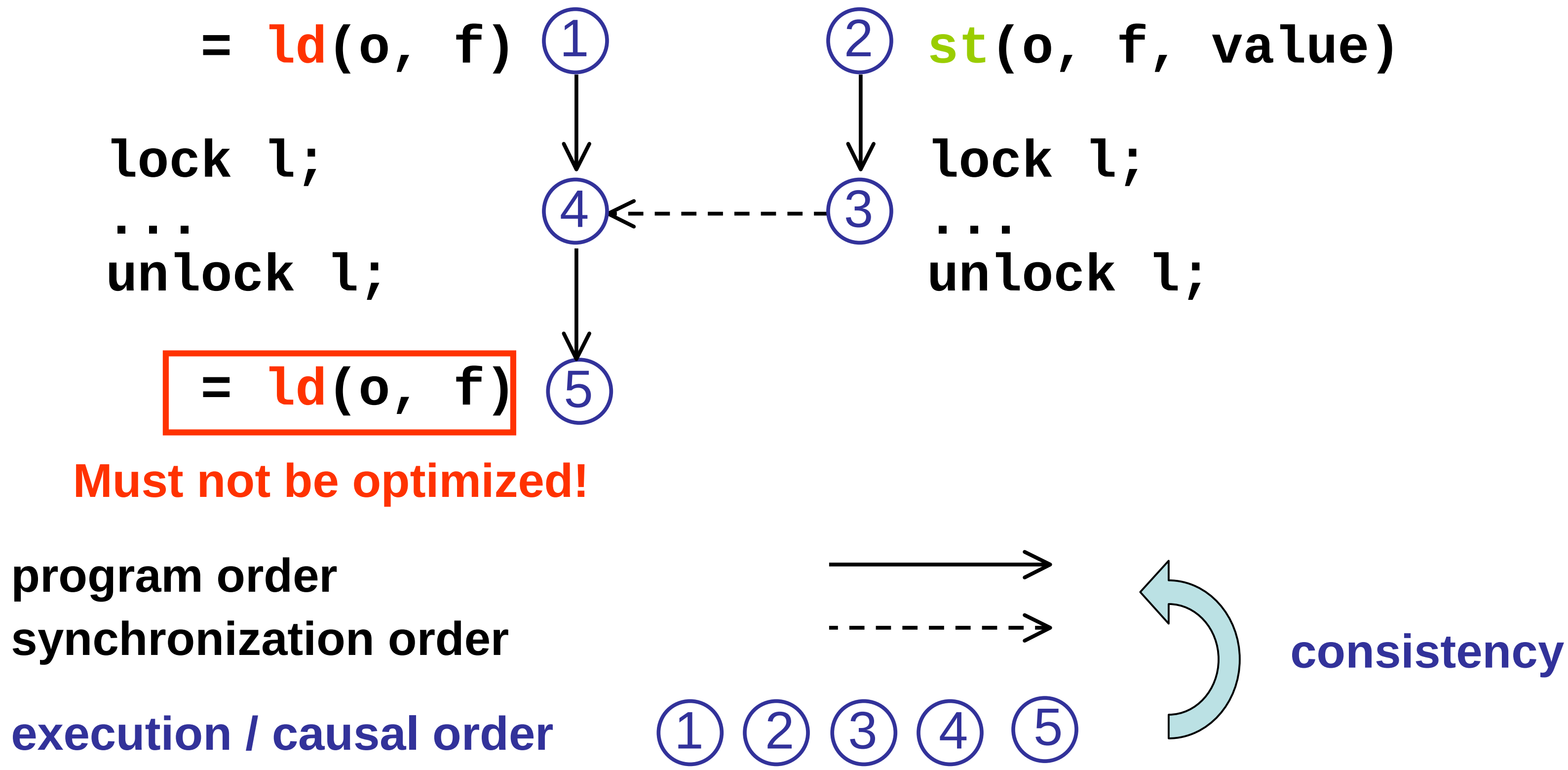
optimized **original**

- SC: {(0,0),(1,0),(1,1)} $\not\subseteq$ (!) {(0,0),(1,1)}
- JC: {(0,0),(1,0),(1,1)} $\subseteq$ {(0,0),(1,0),(1,1)}

- Correctness depends on memory model
- Access to **s1,s2** not "correctly synchronized"

Multi-threading (3/3)

- Synchronization “kills”:



- Similar: access to **`volatile`** variable “kills”.
- Criterion for correct optimization of Java.

2 Strategies …

... to determine the absence of “killing” interference:

Strategy 1: **Synchronization kills**

+ simple, all fields, all accesses treated equally
- only correct for Java Consistency (JC)
- optimization potential not fully exploited

Strategy 2: **Exploit synchronization information**

- Aggressive optimization of thread-local and shared non-conflicting data
- No optimization of shared conflicting data

+ independent of memory model (correct for SC)
- needs concurrency and side-effect info

Procedure

- Whole program analysis
  - Side-effect analysis
  - Conflict analysis (Strategy 2)

- Intra-procedural load-elimination
  - based on SSA-PRE-based [Chow et. al., PLDI 97]
  - lexical equivalence of path expressions
  - Extensions that account for
    - side-effects
    - precise exceptions
    - concurrency (Strategy 2)

Conflict analysis

- Criterions for absence of a conflict?

  1. object is stack/thread-local
  2. accesses between NEW and orderly ESCAPE
  3. accesses before all STARTs
  4. accesses after all JOINs
  5. common protection through a unique lock

- Enhanced and improved version of [PraunGross PLDI03]

Strategy 2: Aggressive optimization

Absence of conflict on object **`o`** and field **`f`** allows for aggressive optimization across synchronization statements:

```
l1 = ld(o,f);
lock l;
...
unlock l;
l2 = ld(o,f);
```

→

```
l1 = ld(o,f);
lock l;
...
unlock l;
l2 = l1;
```

Reasoning:

- If **`o`** is not conflicting, then …
- … lock **`l`** is not involved in protecting **`o`**

Evaluation

- Application and library (GNU 2.96)
- Configurations:

Strategy 1

(orig) no load elimination
(A) basic (call and synchronization kill)
(B) side-effect + synchronization-kills
(C) side-effect + conflict info
(D) side-effect + "perfect" synchronization

Strategy 2

Optimized expressions (compile-time)

	(B) %	(C) %	(D) %
moldyn (*)	109.3	**37.3**	118.0
montecarlo (*)	128.9	142.7	149.1
mtrt (*)	192.0	202.6	210.9
tsp (*)	121.2	127.8	132.2
compress	126.7	146.6	146.6
db	123.1	176.2	176.2
jess	120.6	**184.2**	184.2
avg.	131.7	145.3	159.6

(*) multi-threaded **Strategy 1** **Strategy 2**

Percentage of eliminated expressions
basic configuration (A) = 100%.

Eliminated accesses (runtime)

	(A) %	(B) %	(C) %
moldyn (*)	41.1	41.1	**14.6**
montecarlo (*)	55.6	66.1	70.3
mtrt (*)	0.6	9.1	9.1
tsp (*)	25.6	25.3	25.0
compress	21.5	29.3	30.1
db	11.9	11.9	**32.7**
jess	17.4	17.4	17.8
avg.	23.4	28.6	28.5

(*) multi-threaded **Strategy 1** **Strategy 2**

Percentage of eliminated accesses
un-optimized (orig) = 100%.

Related work

- SSAPRE: Chow et. al. [PLDI 97]
- Load reuse analysis: Bodik et al. [PLDI 99]
- Register promotion by sparse PRE of loads and stores: Lo et al. [PLDI 98]
- Concurrent SSA for SPMD programs: Lee, et. al. [PPoPP 99]
- PRE-based load elimination for Java: Hosking et. al. [SP&E 2001]

Concluding remarks

- Load elimination is effective: up to 55% (avg. 25%) fewer loads than in the original program.

- Side-effect information reduces the number of loads on avg. by another 5%.

- Simple load elimination requires a weak memory model for correctness.

- Accurate information about concurrency can…
  - … make the optimization independent of the MM
  - … enable aggressive opt. across synchronization stmts.

Thank you for your attention.

Eliminated accesses (runtime)

	(orig) 100% mio. accs	(A) %	(B) %	(C) %
moldyn (*)	1651.3	58.9	58.9	**85.4**
montecarlo (*)	478.6	44.4	33.9	29.7
mtrt (*)	366.9	99.4	90.9	90.9
tsp (*)	899.0	74.4	74.7	75.0
compress	2423.5	78.5	70.7	69.9
db	446.6	88.1	88.1	**67.3**
jess	323.5	82.6	82.6	82.2
avg.		76.6	71.4	71.5

(*) multi-threaded **Strategy 1** **Strategy 2**