Static Conflict Analysis for Multi-Threaded Object Oriented Programs

Christoph von Praun
and Thomas Gross

Presented by Andrew Tjang

Authors

- Von Praun
  - Recent PhD
  - Currently at IBM (yorktown heights)
  - Compilers and runtime systems for OOP
- Gross
  - Stanford Graduate
  - CMU faculty
  - Compilers, software construction

Introduction

- How to get info about object use/sharing in multi threaded environments?
- Introducing Object Use Graphs (OUG)
- Provides info to compiler for optimizations
  - o/w compiler makes assumptions
- Check for race conditions in programs (for sw engineers)
- Analysis done at compile time!

Background and Problems

- Escape analysis produce Heap Shape Graphs (HSG)
  - Classifies objects according to properties
- Info valid for *whole* program
  - Accesses in 2 threads cause obj to be shared
- How to get finer granularity?

OUGs

- OUG constructed to determine the “happened before” relationship
- Refines HSG’s escape info
  - Recognizes structural, temporal, and lock protections
- OUG foundation to concurrence aware compiler systems
- Distinguish effects of different threads on abstract objects

Terminology

- Abstract thread, abstract object
  - Compiler entities
  - Conservative approximate runtime entities
- Runtime thread, runtime object
  - Actual objects and threads that exist in a program's execution

Example HSG

```
class Shared {
    int i;
    Shared () { i = 0; }                    // (20)
}

class Example extends Thread {
    static Shared s_field;
    static Object lock_ = new Object();

    static void main(String[] args) {
        Shared s_local = new Shared();      // (2),(3)
        s_local.i++;                        // (4),(5)

        s_field = s_local;                  // (6)
        s_field.i++;                        // (7),(8),(9)

        Thread t = new Example();
        t.start();                          // (10)

        synchronized(lock_) {
            s_field.i++;                    // (11),(12),(13)
        }

        t.join();                           // (14)
        s_field.i++;                        // (15),(16),(17)
    }

    void run() {                            // (22)
        synchronized(lock_) {
            s_field.i++;                    // (23),(24),(25)
        }                                   // (26)
    }
}
```

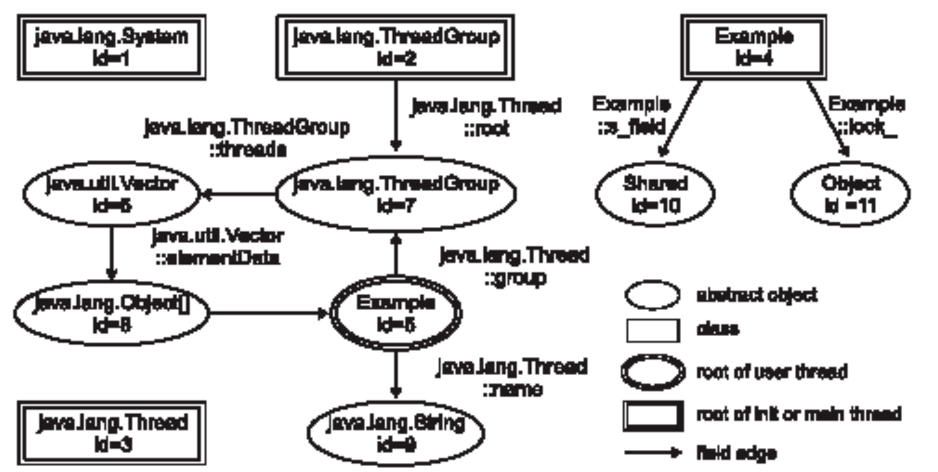

Figure 2: HSG for the example program.

Modeling Object Uses

- Nodes in OUG = events
- Edges = “happened before” relation
- Nodes:
  – Get/Put, Load/Store/Escape, Tstart/Tjoin, Entry/Exit, Call
- Nodes have attributes
  – Abstract thread, program site, host object, accessed field, set of locks, etc…

OUG Edges

- Control-flow ordering
  – Order of events in threads (can be cyclic)
- Reference flow ordering
  – Restriction – object used after object creation or sharing
- Thread-relation ordering
  – Edges to/from TStart/Tjoin nodes

Conflicting vs. Safe

- Objects are conflicting if:
  - No ordering between events
  - Events from different threads
  - At least one event is a Put
  - Accesses are not done under locks
- Safe otherwise
- OUGs may have false positives

OUG Construction

- Determine abstract threads
- Build HSG
- Build OUGs from symbolic execution of abstract threads
- Analyze OUG for conflicting events

Determining Abstract Threads

- Represented by: T := (tid, (m0…mn), kind, multi)
- Tid= unique id for thread
- M0-mn = entry methods for thread
- Kind = (init, main, or user)
- Mult = unique vs multiple
- Threads characterized by methods executed from call graph rooted @ entry methods

Computing the HSG

- Flow insensitive of data and reference relations
- Compositional
  - Methods are analyzed independantly, and summaries used
- Summary has context (parameters return values, etc)
  - Reference vars id'd by alias set
    - AS:=(fieldmap, props, tidmask)
    - Fieldmap = field names to alias set (reachable)
    - Props=properties
    - Tidmask =abstract threads that access

More on HSGs

- Alias sets for class & abstract threads become root nodes
- Reachable from root (transitively) = global => global, could be multi thread accessible
- Method summary
  - MS[m] = ((f0…fn),ret,except,alloc,reads,writes)
  - F0…fn = local variables
  - Collection of alias sets
  - Abstract thread id noted at all object access sites

Symbolic Execution

- Narrows classification into conflicting and non conflicting
- This is where OUGs are constructed
- Maps onto the HSG
- Follows the program execution through the nodes (objects)

MOUGs

- Models relevant events at method level
- Control graph where actions that do not involve the object are pruned
- Created through single flow sensitive method traversal
- Relevant events are local variable l0-ln, or global alias set if class

MOUGs

```
class Shared {
    int i;
    Shared () { i = 0; }                    // (20)
}

class Example extends Thread {
    static Shared s_field;
    static Object lock_ = new Object();

    static void main(String[] args) {
        Shared s_local = new Shared();      // (2),(3)
        s_local.i++;                        // (4),(5)

        s_field = s_local;                  // (6)
        s_field.i++;                        // (7),(8),(9)

        Thread t = new Example();
        t.start();                          // (10)

        synchronized(lock_) {
            s_field.i++;                    // (11),(12),(13)
        }

        t.join();                           // (14)
        s_field.i++;                        // (15),(16),(17)
    }

    void run() {                            // (22)
        synchronized(lock_) {
            s_field.i++;                    // (23),(24),(25)
        }                                   // (26)
    }
}
```

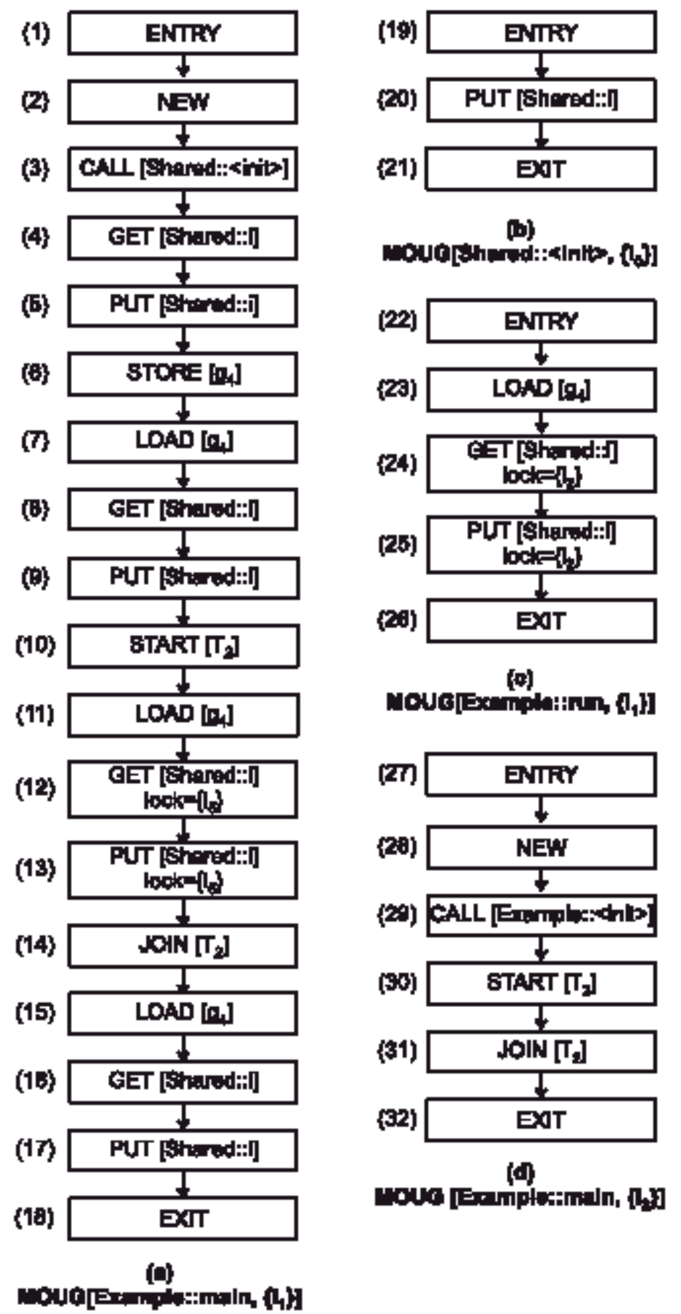

Figure 5: Example MOUGs.

Optimization

- @ Compile time, all execution paths = expensive traversals
- Save some time by avoiding equivalent method, thread, and locking context
  - Use site context SC:=(m,(a0…an),tid,lockset)
- Avoid descent into methods that have no effect on shared data

Conflict Analysis

- Events between new and escape safe
- Events before TSTART are safe if issued by a unique thread?
  - Unless thread is started multiple times
- If only Get events left, obj is read-only
- Else check if lock protected
- Else check readonlyness/lock protected of individual fields

Benchmarks

	philo	elevator	mtrt	sor	tsp	hedc	mold	ray	monte
program characteristics									
appl loc	81	528	11298	300	706	28299	1402	1972	3674
appl classes	2	5	34	7	4	48	11	19	19
lib classes	129	142	158	132	141	208	129	131	146
methods in call graph	192	311	722	205	302	1025	224	270	441
bytecodes in call graph	3605	6820	20137	4483	6481	24375	6531	5982	8161
user threads	2	2	2	3	2	5	2	2	2
method spec	68	118	578	16	108	3653	111	150	267
compilation resources									
shape analysis [s]	0.7	1.3	2.6	0.7	1.6	6.5	0.9	0.9	1.1
symb exec [s]	0.5	0.8	2.5	0.5	0.8	123.6	0.9	0.8	1.3
meth sites proc	103	191	1090	50	168	29254	156	209	431
meth sites reused	85	163	1640	43	123	60233	244	285	452
meth sites noeffect	100	179	855	81	163	29423	136	174	358
conflict analysis [s]	0.1	0.2	2.8	0.1	0.2	433.8	0.9	0.9	0.5
memory [MB]	0.5	3.0	14.7	0.5	1.5	263.5	1.5	3.4	3.8

Table 1: Benchmark characterization and compilation properties.

Characteristics

	philo	elevator	mtrt	sor	tsp	hedc	mol	ray	monte
classification of HSG nodes									
class	131	147	192	139	145	256	140	150	165
inst	43	65	199	44	62	467	51	71	79
inst unique	29	43	122	31	39	356	33	35	51
shared	10	13	97	3	13	184	16	29	36
shared readonly	3	6	55	1	6	116	6	12	28
shared lock-protected	6	3	36	1	4	30	6	6	2
shared mix-protected	0	0	1	0	0	2	0	3	1
shared conflicting	1	4	5	1	3	36	4	8	5
OUGs									
nodes max	217	327	1618	286	311	83052	537	302	726
nodes median	50	95	74	116	99	417	99	64	59
edges max	435	689	6083	410	640	206456	616	616	2450
edges median	67	172	111	221	163	748	145	92	84

Table 2: Characteristics of HSG and OUGs (no arrays).

How well does it do?

	philo	elevator	mtrt	sor	tsp	hedc	mol	ray	monte
global									
abstract objects	13	38	91	14	30	201	25	39	54
allocation sites	18	55	117	18	42	256	27	56	59
access sites	135	526	1002	288	478	1954	963	466	399
r/w shared									
abstract objects	7	10	59	5	9	107	14	24	20
allocation sites	12	17	89	4	13	180	19	43	29
access sites	111	246	956	197	337	1818	899	408	252
OUG (lock protection)									
abstract objects	2	7	8	5	6	76	8	18	20
allocation sites	2	9	19	4	5	163	7	31	29
access sites	21	168	165	155	190	1387	751	254	216
OUG (all)									
abstract objects	1	4	5	1	3	36	4	8	5
... improvement (%)	86	60	91	80	67	63	71	67	75
allocation sites	1	6	16	2	3	129	5	16	15
... improvement (%)	92	65	82	50	77	28	74	63	48
access sites	11	113	121	75	58	1110	529	144	118
... improvement (%)	90	54	87	62	83	38	41	65	53
avg/max alloc sites per obj.	1.0/1	1.5/2	4.0/9	2.0/2	1.0/1	4.1/63	1.3/2	2.0/4	3.8/9
conflicting fields	2	12	20	11	6	198	50	19	19
avg/max acc sites per field	5.5/8	9.3/29	5.7/23	6.8/11	9.7/14	4.8/33	10.6/127	5.5/13	5.9/23
conflict types									
all writes locked	0	2	1	0	2	11	0	0	2
object local to thread	1	1	1	1	1	2	1	1	0
one lock but not unique	0	1	2	0	0	8	0	0	2
no common lock	0	0	1	0	0	15	3	7	1

Table 3: Static conflict detection (no arrays).

How useful?

	philo	elevator	mtrt	sor	tsp	hedc	mol	ray	monte
shared									
allocated	11	43	440	4	10011	861	2064	2103951	20020
actually shared	8	37	15	4	375	207	5	345	20013
conflict									
allocated	2	33	6	2	5002	491	2051	2103667	20007
actually conflict	0	0	1	0	163	15	1	69	1

Table 4: Allocation of objects with their compile-time classification and the actual situation at runtime.

How fast?

	mtrt	sor	tsp	mol	ray	monte
no instrumentation						
base	20.8	3.8	8.9	20.6	49.4	23.4
optimized[1]	19.9	3.2	8.9	—	46.1	22.6
object race checking						
stack-escape	41.7	3.9	23.8	64.0	116.1	41.5
global	29.6	3.9	23.8	65.5	111.5	41.3
shared r/w	29.0	3.9	23.9	65.4	110.9	42.0
OUG	28.5 (37%)	3.9 (3%)	10.3 (16%)	66.0 (220%)	82.7 (67%)	40.9 (75%)
OUG optimized	27.7 (39%)	3.3 (3%)	10.1 (14%)	38.4 (86%)	73.0 (58%)	40.4 (79%)

Table 7: Runtime in seconds and overhead of the program instrumentation (array access not instrumented).

Limitations

- Initializer and finalizer threads not considered
- Whole Program Knowledge
  - Reflection/dynamic class loading
- Naïve (conservative) thread ordering assumptions

Conclusion

- OUGs can provide finer grained analysis of shared objects
- Fewer accesses are classified as conflicting (incorrectly)
- OUGs solid foundation for reporting accesss conflicts and optimize synchronization operation placement