CCC Seminar

Composable, Nestable, Pessimistic Atomic Statements

OOPSLA '11 Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications

Benjamin Weber

Authors

Zachary Anderson

ETH Zürich

David Gay

Intel Labs Berkeley

Introduction

- Few extensions to the C programming language → “shelters”
- Compiler pass transforms “shelter code” to C code with calls to shelter runtime
- Shelter runtime ensures atomicity (and other properties) given correct calls to the runtime

Shelter code elements

- Shelter type: shelter_t
- Type annotation: sheltered_by(…)
- Atomic block: atomic { … }
- Function annotation: needs_shelters(…)

→ “relatively” small annotation overhead

shelter_t type

Normal C type, no special restrictions

```
shelter_t shelter_variable;
struct some_struct {
    shelter_t shelter_field;
    int other_struct_field;
};
shelter_t some_function(shelter_t param);
```

sheltered_by type annotation

Annotation for shared objects

```
shelter_t shelter_variable;
int sheltered_by(shelter_variable) some_int;
struct some_struct {
    shelter_t shelter_field;
    int sheltered_by(shelter_field) other_struct_field;
};
```

atomic block

```
atomic {
  /*
   guarantees atomic access to sheltered objects
   needs correct annotations
   open_atomic { … } & force_open_atomic { … } for open nesting
   */
}
```

needs_shelters function annotation

- Required to avoid whole-program analysis
- Somewhat complicated
- Not important for understanding of the idea
- See Appendix

Example (from paper)

```
typedef struct {
    int sheltered_by(s) id;
    float sheltered_by(s) balance;
    shelter_t s;
} account_t;
```

Example (from paper)

```
needs_shelters(a->s)
void deposit(account_t* a, float d) {
   a->balance += d; // not atomic, see next slide
}
needs_shelters(a->s)
void withdraw(account_t* a, float d) {
   a->balance -= d; // not atomic, see next slide
}
```

Example (from paper)

```
needs_shelters(to->s, from->s)
void transfer(account_t* to, account_t* from, float amount) {
   atomic {
      // here accesses become atomic
      withdraw(from, amount);
      deposit(to, amount);
   }
}
```

Implementation

- No whole-program analysis required
- Supports explicit external locks
  - Through shadow shelters (→ more annotations)
- Supports condition variables (→ more annotations)
- Supports both open- and closed-nesting
  - Closed-nesting: Changes become visible at the end of outer-most atomic block
  - Open-nesting: Changes become visible at the end of each nested atomic block resp.

Implementation

- Timestamp based (similar to database transactions)
  - Global counter → contention → exponential back-off
- Pessimistic: First makes sure it's safe to execute atomic blocks, then executes them
- Not Optimistic: Execute code and if a problem is detected roll-back changes (roll-back may be expensive or impossible e.g. for IO)
- Must know used shelters before atomic block
  - For struct fields program analysis may be imprecise (see appendix)
    - Each struct with shelters has its own global shelter which can be used for this case → quite extreme (problematic for the sqlite benchmark)
    - Could use more fine-grained shelter hierarchy → might require whole-program analysis

Formalism

- Paper introduces formalism for shelter semantics
- Operational semantics
- Rather complicated (see appendix & paper)
- Allows to formally establish useful properties about shelters
  - Deadlock freedom
  - Partial atomicity for sheltered objects
  - No guarantees about starvation or fairness

Benchmarks

- Benchmarked with 13 different programs
- Including
  - SQLite database system
  - parallel bzip2 (pbzip2)
  - n-body simulation (ebarnes)
  - oatomic (using open nesting)
- Executed on 2.27GHz Intel Xeon X7560 with four processors each with eight cores (total 32 cores) with 32GB memory without Hyperthreading

Benchmark

- Compared against
  - explicit locking (reference)
  - Autolocker
  - Intel C/C++ compiler software transactional memory
  - Single global lock
  - Shelters implemented using RWLocks

Benchmark

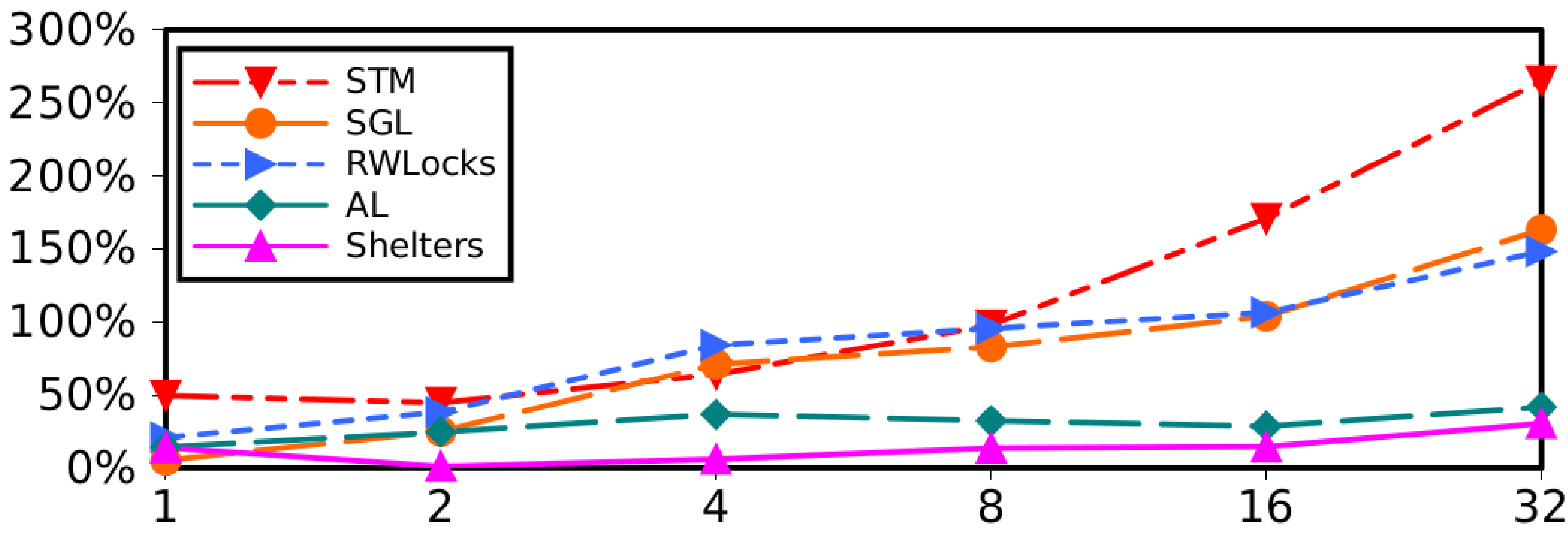

Figure 5. Average percent slowdown with respect to explicit locking over all benchmarks versus the number of threads. Lower is better.

Questions?

Appendix

needs_shelters function annotation

- Required if the function is called inside an atomic block
- Must declare which shelters are used inside function
- For calls to other functions, must also declare their used shelters
- Can use globals & parameter expressions
- Compiler & runtime give errors if they are missing
- Missing annotations can lead to data-races, but not deadlocks

needs_shelters function annotation

```
needs_shelters(shelter_variable)
void some_function() { … }

needs_shelters(arg->shelter_field)
void another_function(struct some_struct arg)
{ … }
```

- needs_shelters is a var arg function

Example (from Paper)

```
void idTransfer(int toId, int fromId, float a) {
   // this example will require the global account_t shelter
   atomic {
      account_t *to = accountLookup(toId);
      account_t *from = accountLookup(fromId);
      withdraw(from, a);
      deposit(to, a);
   }
}
```

Example (from Paper)

```
open_atomic {
  for (t = l->head; t; t = t->next) {
    atomic {
      withdraw(t->from, a);
      deposit(t->to, a);
} } }
```

Formalism: Definitions

Declaration	d	$::=$	int v sheltered by x
Trace	T	$::=$	$(t_1, s_1), \ldots, (t_m, s_m)$
Statement	s	$::=$	reserve$(\sigma_1, \ldots, \sigma_m)$
		$\mid$	register$(\sigma_1, \ldots, \sigma_m)$
		$\mid$	pop
		$\mid$	$v := v_1 + v_2 + n$
Shelter (Σ)	σ	$::=$	$v_\sigma \mid x$

Identifiers	v, x	Integers	t, n, m

Figure 2. Traces of shelter-based programs.

Formalism: Rules

$$\frac{\text{regfor}(H,t,v) \qquad \text{regfor}(H,t,v_1) \qquad \text{regfor}(H,t,v_2)}{R,H \models (t, v := v_1 + v_2 + n)}$$

$$\frac{\text{access}(H,t,v) \qquad \text{access}(H,t,v_1) \qquad \text{access}(H,t,v_2)}{\begin{array}{c} M,R,H,a : (t, v := v_1 + v_2 + n) \rightarrow \\ M[v \rightarrow M(v_1) + M(v_2) + n], R, H, a \end{array}}$$

$$\frac{H(t) = \emptyset \vee \text{subsumed}(\{\sigma_1, \ldots, \sigma_m\}, R(t))}{R,H \models (t, \text{reserve}(\sigma_1, \ldots, \sigma_m))}$$

$$\frac{R' = R[t \rightarrow \{\sigma_1, \ldots, \sigma_m\}]}{M,R,H,a : (t, \text{reserve}(\sigma_1, \ldots, \sigma_m)) \rightarrow M, R', H, a}$$

$$\frac{\text{subsumed}(\{\sigma_1, \ldots, \sigma_m\}, R(t)) \qquad m \geq 1}{R,H \models (t, \text{register}(\sigma_1, \ldots, \sigma_m))}$$

$$\frac{\begin{array}{c} H' = H[t \rightarrow H(t) \cup \{(a, \sigma_1), \ldots, (a, \sigma_m)\}] \\ \neg\text{cycle}(\text{impedes}(R, H')) \end{array}}{M,R,H,a : (t, \text{register}(\sigma_1, \ldots, \sigma_m)) \rightarrow M, R, H', a+1}$$

$$\frac{H(t) \neq \emptyset}{R,H \models (t, \text{pop})}$$

$$\frac{\begin{array}{c} b = \max\ \{a \mid (a,\sigma) \in H(t)\} \\ H' = H[t \rightarrow \{(a,\sigma) \mid (a,\sigma) \in H(t) \wedge a \neq b\}] \end{array}}{M,R,H,a : (t, \text{pop}) \rightarrow M, R, H', a}$$

$$\frac{R,H \not\models (t_1, s_1)}{M,R,H,a : (t_1,s_1), \ldots, (t_m, s_m) \rightarrow \bot, \bot, \bot, \bot : \epsilon}$$

$$\frac{R,H \models (t_1, s_1) \qquad M,R,H,a : (t_1, s_1) \rightarrow M', R', H', a'}{\begin{array}{c} M,R,H,a : (t_1,s_1), \ldots, (t_m,s_m) \rightarrow \\ M', R', H', a' : (t_2, s_2), \ldots, (t_m, s_m) \end{array}}$$

$$\begin{array}{rcl} \text{regfor}(H,t,v) & = & \exists (a,\sigma) \in H(t). v_\sigma \leq \sigma \\ \text{interferes}(\sigma_1, \sigma_2) & = & \sigma_1 \leq \sigma_2 \vee \sigma_2 \leq \sigma_1 \\ \text{access}(H,t,v) & = & \exists (a,\sigma) \in H(t).(v_\sigma \leq \sigma \wedge \forall t' \neq t. \forall (a', \sigma') \in H(t').\text{interferes}(v_\sigma, \sigma') \Rightarrow a < a') \\ \text{impedes}(R,H)(t_1,t_2) & = & \exists (\sigma_1, a_1) \in H(t_1).(\exists (\sigma_2, a_2) \in H(t_2).a_1 < a_2 \wedge \text{interferes}(\sigma_1, \sigma_2) \vee \\ & & \quad \exists \sigma_2 \in R(t_2). \wedge \text{interferes}(\sigma_1, \sigma_2)) \end{array}$$

$$\text{subsumed}(\Sigma_1, \Sigma_2) = \forall \sigma_1 \in \Sigma_1. \exists \sigma_2 \in \Sigma_2. \sigma_1 \leq \sigma_2$$

Figure 3. Trace Operational Semantics